

Microsoft®

Microsoft®

5.0

MICROSOFT® C OPTIMIZING COMPILER

FOR THE MS-DOS® OPERATING SYSTEM

USER'S GUIDE

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of the agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose other than the purchaser's personal use without the written permission of Microsoft Corporation.

© Copyright Microsoft Corporation, 1984-1987. All rights reserved. Simultaneously published in the U.S. and Canada.

If you have comments about the software, complete the Product Assistance Request form at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback card at the back of this manual and return it to Microsoft Corporation.

Microsoft®, MS®, MS-DOS®, CodeView®, and XENIX® are registered trademarks and QuickC™ is a trademark of Microsoft Corporation.

AT&T® is a registered trademark of American Telephone & Telegraph Company.

DEC®, PDP®, and VAX® are registered trademarks of the Digital Equipment Corporation.

IBM® is a registered trademark of the International Business Machines Corporation.

Intel® is a registered trademark of Intel Corporation.

Olivetti® is a registered trademark of Olivetti SpA.

Texas Instruments® is a registered trademark of the Texas Instruments Corporation.

Turbo Pascal™ is a trademark of Borland International, Inc.

UNIX® is a registered trademark of AT&T Bell Laboratories.

Wang® is a registered trademark of Wang Laboratories Incorporated.

Z8000® is a registered trademark of Zilog, Inc.

TABLE OF CONTENTS

1	Introduction.....	1
1.1	Overview.....	3
1.2	About This Manual.....	4
1.3	New Features.....	6
1.4	Notational Conventions.....	8
1.5	Books about C	11
1.6	Requesting Assistance.....	11
2	Getting Started.....	13
2.1	Introduction.....	15
2.2	Backing Up Your Disks.....	15
2.3	Disk Contents	15
2.4	The SETUP Program.....	16
2.4.1	What SETUP Does	16
2.4.2	Running SETUP.....	17
2.4.2.1	Choosing SETUP Command-Line Options	18
2.4.2.2	Choosing Floating-Point Math Packages.....	20
2.4.2.3	Adding Graphics to Combined Libraries	21
2.4.2.4	Building Libraries	22
2.4.2.5	Compiling Programs after Installation.....	23
2.4.3	Library-Naming Conventions.....	23
2.4.4	How SETUP Organizes Files	24
2.4.5	Setting Up the Environment	26

CONTENTS

2.4.6	Using Uncombined Libraries	26
2.4.7	Using the Microsoft C Graphics Library	28
2.5	Understanding the Compiler Software	28
2.5.1	Executable Files	28
2.5.2	Include Files.....	29
2.5.3	Library Files.....	30
2.5.4	Other Files.....	30
2.5.5	The Compiler Environment	32
2.5.6	Environment Variables.....	33
2.5.6.1	The PATH Variable	34
2.5.6.2	The LIB Variable	35
2.5.6.3	The INCLUDE Variable	35
2.5.6.4	The TMP Variable	35
2.5.6.5	The CL Variable.....	35
2.5.6.6	Defining Environment Variables.....	36
2.5.6.7	Environment Variables and CL Options.....	37
2.5.7	The CONFIG.SYS File.....	37
2.6	Using a RAM Disk	38
2.7	Converting Existing C Programs.....	38
2.8	Sample Compiler Command Lines	39
2.8.1	Listing CL Options	39
2.8.2	Simple Compile and Link	40
2.8.3	Using Wild-Card Characters	40
2.8.4	Compiling without Linking.....	40
2.8.5	Using the Alternate Math Library	40
2.8.6	Preparing to Use the CodeView Debugger	41
2.8.7	Setting Titles and Subtitles	41
2.9	Practice Session	41
3	Compiling with the CL Command.....	45
3.1	Introduction.....	47

3.2	The Basics:	
	Compiling and Linking C Programs.....	48
3.2.1	The CL Command	48
3.2.1.1	Specifying Source and Object Files	48
3.2.1.2	Creating Executable Files	50
3.2.1.3	Stopping CL	51
3.2.2	Using the CL Environment Variable	51
3.3	Using CL Options	53
3.3.1	Memory-Model (/A) and Floating-Point (/FP) Options	54
3.3.2	Listing the Compiler Options (/HELP)	56
3.3.3	Specifying Source Files (/Tc)	56
3.3.4	Compiling without Linking (/c)	57
3.3.5	Compiling with QuickC (/qc, /Zr, /Zq).....	57
3.3.6	Naming the Object File (/Fo)	58
3.3.7	Naming the Executable File (/Fe)	60
3.3.8	Creating Listing Files	61
3.3.8.1	Types of Listings (/Fs, /Fl, /Fa, /Fc, /Fm)	61
3.3.8.2	Special File Names	64
3.3.8.3	Setting Line Width (/Sl) and Page Length (/Sp)	65
3.3.8.4	Setting Titles (/St) and Subtitles (/Ss)	66
3.3.8.5	Formats for Listings	67
3.3.9	Controlling the Preprocessor	75
3.3.9.1	Defining Constants and Macros (/D)	75
3.3.9.2	Predefined Identifiers	77
3.3.9.3	Removing Definitions of Predefined Identifiers (/U, /u).....	78
3.3.9.4	Producing a Preprocessed Listing (/P, /E, /EP)	79
3.3.9.5	Preserving Comments (/C).....	80
3.3.9.6	Searching for Include Files (/I, /X)	80
3.3.10	Using the 80186, 80188, or 80286 Processor (/G0, /G1, /G2)	81
3.3.11	Checking for Program Errors	82
3.3.11.1	Understanding Error Messages.....	82
3.3.11.2	Setting the Warning Level (/W, /w).....	84
3.3.11.3	Checking Syntax (/Zs)	85
3.3.11.4	Generating Function Declarations (/Zg)....	86

3.3.12	Preparing for Debugging (/Zi, /Zd, /Od)	87
3.3.13	Optimizing	88
3.3.13.1	Controlling Optimization (/O Options)	89
3.3.13.2	Removing Stack Probes (/Gs)	97
3.3.14	Enabling and Disabling Language Extensions (/Ze, /Za)	99
3.3.15	Packing Structure Members (/Zp)	100
3.3.16	Setting the Stack Size (/F)	102
3.3.17	Restricting the Length of External Names (/H)	103
3.3.18	Labeling the Object File (/V)	103
3.3.19	Suppressing Default-Library Selection (/Zl)	104
3.3.20	Changing the Default char Type (/J)	105
3.3.21	Controlling Stack and Heap Allocation	105
3.3.22	Controlling the Calling Convention (/Gc)	106
3.3.23	Compiling for Windows Applications (/Aw, /Gw)	109
3.3.24	XENIX-Compatible Options	109
3.4	Controlling Binary and Text Modes	111
4	Linking with the CL Command	113
4.1	Introduction	115
4.2	The Default Linking Process	115
4.3	Passing Linker Information: The /link Option	115
4.3.1	Specifying Libraries	116
4.3.1.1	Linking with Additional Libraries	117
4.3.1.2	Looking in Different Locations for Libraries	117
4.3.1.3	Overriding Libraries Named in Object Files	117
4.3.2	Specifying Linker Options	119
4.3.2.1	Defining Linker Options on the CL Command Line	119
4.3.2.2	Defining Linker Options in the Environment	120

4.4	Linker Options	120
5	Running C Programs on MS-DOS	125
5.1	Introduction	127
5.2	Passing Command-Line Data to a Program	127
5.2.1	Expanding Wild-Card Arguments	130
5.2.2	Suppressing Command-Line Processing	131
5.3	Returning an Exit Code	131
5.4	Suppressing Null-Pointer Checks	132
6	Working with Memory Models	135
6.1	Introduction	137
6.2	Near, Far, and Huge Addressing	137
6.3	Using the Standard Memory Models	139
6.3.1	Creating Small-Model Programs	140
6.3.2	Creating Medium-Model Programs	141
6.3.3	Creating Compact-Model Programs	141
6.3.4	Creating Large-Model Programs	142
6.3.5	Creating Huge-Model Programs	143
6.4	Using the near, far, and huge Keywords	144
6.4.1	Library Support for near, far, and huge	145
6.4.2	Declaring Data with near, far, and huge	146
6.4.3	Declaring Functions with the near and far Keywords	148
6.4.4	Pointer Conversions	150
6.5	Creating Customized Memory Models	152
6.5.1	Code Pointers	153
6.5.2	Data Pointers	153
6.5.3	Setting Up Segments	154
6.5.4	Library Support for Customized Memory Models	155
6.6	Setting the Data Threshold	156

6.7	Naming Modules and Segments.....	157
6.8	Specifying Text and Data Segments	159
7	Controlling Floating-Point Math Operations	161
7.1	Introduction	163
7.2	Summary of Math Packages.....	163
7.2.1	The Emulator Package	163
7.2.2	The 8087/80287 Package	164
7.2.3	The Alternate Math Package	164
7.3	Selecting Floating-Point (/FP) Options	165
7.3.1	The /FPi Option.....	167
7.3.2	The /FPi87 Option	168
7.3.3	The /FPc Option	168
7.3.4	The /FPc87 Option.....	169
7.3.5	The /FPa Option.....	169
7.4	Library Considerations for Floating-Point Options.....	170
7.4.1	In-Line Instructions or Calls.....	170
7.4.2	Using One Standard Library for Linking	170
7.5	Compatibility between Floating-Point Options	173
7.6	Using the NO87 Environment Variable.....	174
7.7	If Your Computer Is Not IBM Compatible	175
8	Improving Program Speed.....	177
8.1	Introduction	179
8.2	Using Register Variables	179
8.3	Optimization Options and Pragmas	181
8.3.1	Default Optimization	181
8.3.2	Generating Intrinsic Functions.....	181

8.3.3	Relaxing Alias Checking	182
8.3.4	Performing Loop Optimizations.....	182
8.3.5	Removing Stack Probes	183
8.3.6	Maximum Optimization.....	183
8.4	Choosing the Function-Calling Convention	183
8.5	Efficiency in Large Data Models	184
8.5.1	Changing Addressing with near, far, and huge Keywords.....	184
8.5.2	Setting the Data Threshold.....	185
8.5.3	Controlling Segments Used for Allocation	185
8.6	Efficiency in Large Code Models.....	185

Appendixes

A	Using Exit Codes	189
A.1	Introduction.....	191
A.2	Exit Codes with MS-DOS Batch Files	191
A.3	Compiler Exit Codes	192
B	Converting from Previous Versions of the Compiler	193
B.1	Introduction.....	195
B.2	Differences between Versions 5.0 and 4.0.....	195
B.2.1	Enhancements and Additions.....	195
B.2.2	Changes to the Language Syntax.....	196
B.2.3	New Features for the MS-DOS Implementation of C	198
B.2.4	Changed Library Routines	199
B.2.4.1	Graphics Routines.....	199
B.2.4.2	Heap-Checking Functions	199
B.2.4.3	DOS and BIOS Interface Functions	200

	B.2.4.4	Other New Functions	200
	B.2.4.5	New Include Files	201
B.3		Differences between Versions 4.0 and 3.0.....	203
	B.3.1	Enhancements and Additions	203
	B.3.2	Changes in the Language Syntax.....	204
	B.3.3	New Features for the MS-DOS Implementation of C	206
	B.3.4	New Library Routines and Include Files.....	207
	B.3.5	Changes in Library-Function Syntax	208
C		Writing Portable Programs	209
C.1		Introduction	211
C.2		Program Portability	212
C.3		Machine Hardware	212
	C.3.1	Byte Length	212
	C.3.2	Word Length	212
	C.3.3	Storage Alignment	213
	C.3.4	Byte Order in a Word.....	214
	C.3.5	Bit Fields	215
	C.3.6	Pointers	216
	C.3.7	Address Space	217
	C.3.8	Character Set	217
C.4		Compiler Differences	218
	C.4.1	Signed/Unsigned char and Sign Extension	218
	C.4.2	Shift Operations	218
	C.4.3	Identifier Length	219
	C.4.4	Register Variables	219
	C.4.5	Type Conversion.....	220
	C.4.6	Functions with a Variable Number of Arguments	221
	C.4.7	Side Effects and Evaluation Order	221
C.5		Environment Differences	222
C.6		Portability of Data.....	223
C.7		Type-Size Summary	223
C.8		Byte-Ordering Summary	225

D	Writing Programs for Read-Only Memory	227
D.1	Introduction	229
D.2	MS-DOS-Dependent Library Routines	229
D.3	Floating-Point Math Support.....	230
D.4	Modifying Start-Up Code.....	231
E	Error Messages.....	235
E.1	Introduction	237
E.2	Command-Line Error Messages	237
E.2.1	Command-Line Fatal-Error Messages	238
E.2.2	Command-Line Error Messages.....	238
E.2.3	Command-Line Warning Messages.....	241
E.3	Compiler Error Messages.....	243
E.3.1	Fatal-Error Messages	244
E.3.2	Compilation-Error Messages.....	251
E.3.3	Warning Messages.....	269
E.3.4	Compiler Limits	280
E.4	Run-Time Error Messages	281
E.4.1	Run-Time-Library Error Messages	281
E.4.2	Floating-Point Exceptions	284
E.4.3	Run-Time Limits.....	286
	Glossary	287
	Index	301

Tables

Table 2.1	Default Environment Settings.....	26
Table 3.1	CL Options and Default Libraries.....	55
Table 3.2	Default File Names and Extensions.....	62
Table 3.3	Arguments to Listing Options.....	63
Table 3.4	Using the loop_opt Pragma.....	95
Table 3.5	Using the check_stack Pragma.....	98
Table 3.6	Using the pack Pragma.....	101
Table 3.7	XENIX Options Accepted by the CL Command.....	110
Table 5.1	Argument Variables	128
Table 6.1	Addressing of Code and Data Declared with near, far, and huge	144
Table 6.2	Start-Up Routines for Customized Memory Models	156
Table 6.3	Segment-Naming Conventions	158
Table 7.1	Summary of Floating-Point Options	166
Table C.1	C Type Sizes	224
Table C.2	Byte Ordering for Short Types.....	225
Table C.3	Byte Ordering for Long Types.....	225
Table D.1	MS-DOS-Dependent Library Routines	229
Table E.1	Limits Imposed by the C Compiler.....	280
Table E.2	Program Limits at Run Time	286

CHAPTER

1

INTRODUCTION

1.1	Overview	3
1.2	About This Manual	4
1.3	New Features	6
1.4	Notational Conventions.....	8
1.5	Books about C.....	11
1.6	Requesting Assistance.....	11

1.1 Overview

The C language is a powerful general-purpose programming language that can generate efficient, compact, and portable code. The Microsoft® C Optimizing Compiler for the MS-DOS® operating system is a full implementation of the C language as defined by its authors, Brian W. Kernighan and Dennis M. Ritchie, in *The C Programming Language*. Microsoft Corporation is actively involved in the development of the ANSI (American National Standards Institute) standard for the C language; this version of Microsoft C anticipates and conforms to the forthcoming standard in many areas.

Microsoft C offers several important features to help you increase the efficiency of your C programs. You can choose between five standard memory models (small, medium, compact, large, and huge) to set up the combination of data and code storage that best suits your program. For flexibility and even greater efficiency, the Microsoft C Optimizing Compiler allows you to “mix” memory models by using special declarations in your program.

The C language itself does not provide such standard features as input and output capabilities and string-manipulation features. These capabilities are provided as part of the run-time library of functions that accompanies the Microsoft C Optimizing Compiler. Because the functions that require interaction with the operating system (for example, input and output) are logically separate from the language itself, the C language is especially suited for producing portable code.

The portability of your Microsoft C programs is increased by the use of a common run-time library for MS-DOS and XENIX® installations. Using the routines in this library, you can transport programs easily from a XENIX development environment to an MS-DOS machine, or vice versa. See Appendix B of the *Microsoft C Run-Time Library Reference* (included in this package) for more information on the common library for MS-DOS and XENIX.

Compared with other programming languages, C is extremely flexible concerning data conversions and nonstandard constructions. The Microsoft C Optimizing Compiler offers several levels of warnings to help you control this flexibility; programs in an early stage of development can be processed using the full warning capabilities of the compiler to catch mistakes and unintentional data conversions. The experienced C programmer can use a lower warning level for programs that contain intentionally nonstandard constructions. See Section 3.3.11.2 for more information about this feature.

1.2 About This Manual

This manual explains how to use the Microsoft C Optimizing Compiler to compile, link, and run C programs on your MS-DOS system. The manual assumes that you are familiar with the C language and with MS-DOS, and that you know how to create and edit a C-language source file on your system.

Note

Since MS-DOS and PC-DOS are essentially the same operating system, Microsoft manuals use the term MS-DOS to refer to both systems, except in those cases where the distinction is significant.

If you have questions about the C language, turn to the *Microsoft C Quick Reference Guide* included in this package. The *Microsoft C Run-Time Library Reference* documents the run-time library routines you can use in your C programs. The Microsoft CodeView and Utilities manual explains how to use the CodeView™ symbolic debugger and the other utilities provided in the Microsoft C Optimizing Compiler package. The *Microsoft Mixed-Language Programming Guide* explains how to mix modules written in Microsoft C, Microsoft FORTRAN, Microsoft Pascal, and Microsoft BASIC. For more information about programming in the C language, refer to Section 1.5, "Books about C."

The following list gives brief descriptions of the remaining chapters of the *Microsoft C Optimizing Compiler User's Guide*:

Chapter 2, "Getting Started," covers installation and organization of the compiler software. This chapter explains how to set up an operating environment for the compiler by defining environment variables, and includes a practice session to acquaint you with the Microsoft C Optimizing Compiler.

Chapter 3, "Compiling with the CL Command," discusses the process of compiling a program using the **CL** compiler driver. This chapter describes the options most commonly used to control preprocessing, compiling, and output of files.

Chapter 4, "Linking with the CL Command," describes how to link object files using the **CL** command. This chapter explains how the linker searches for libraries, shows how to specify libraries for linking, and describes the linker options that can be used for C programs.

Chapter 5, "Running C Programs on MS-DOS," explains how to run your executable program file, and discusses features specific to the MS-DOS implementation of C. The chapter tells how to pass data from MS-DOS to a program at execution time, and how to return an exit code from your program to MS-DOS.

Chapter 6, "Working with Memory Models," describes methods of managing memory models. These methods are useful for writing large programs that use more than 64K (kilobytes) of code or data. This chapter also discusses "mixed-model" programming (combining features from the five standard memory models).

Chapter 7, "Controlling Floating-Point Math Operations," describes the options of the **CL** command that control how Microsoft C programs handle floating-point math and the libraries that support it.

Chapter 8, "Improving Program Speed," gives suggestions and hints for maximizing program speed.

Appendix A, "Using Exit Codes," lists the exit codes produced by the Microsoft C Optimizing Compiler. The chapter also briefly discusses how exit codes are used in description files for the **MAKE** program maintenance utility and in batch files.

Appendix B, "Converting from Previous Versions of the Compiler," summarizes the differences between Version 5.0 of the Microsoft C Optimizing Compiler and previous versions. This appendix gives instructions for converting programs written for versions prior to 5.0 to the format accepted by Version 5.0.

Appendix C, "Writing Portable Programs," lists some of the C language features that are implementation dependent, and offers suggestions for increasing program portability.

Appendix D, "Writing Programs for Read-Only Memory," gives information about modifying start-up code and initializing floating-point support for programs that will be put in read-only memory.

Appendix E, "Error Messages," lists and describes the error messages generated by the Microsoft C Optimizing Compiler and by the **CL** command. It also lists and explains run-time error messages produced by executable programs written in C.

1.3 New Features

Several useful new features have been added to Version 5.0 of the Microsoft C Optimizing Compiler. This section summarizes features added since Version 4.0. For information about differences between Version 5.0 and versions prior to 4.0, see Appendix B, "Converting from Previous Versions of the Compiler."

The new features include the following:

Feature	Description																
Microsoft QuickC Compiler	The Microsoft QuickC Compiler is bundled with Version 5.0 of the Microsoft C Optimizing Compiler. The Microsoft QuickC Compiler provides an integrated programming environment including program editor, compiler, debugger, and integrated program- and library-maintenance facilities.																
SETUP program	Batch files to automate installation of the Microsoft C Optimizing Compiler.																
Combined run-time libraries	Combined run-time libraries built by the installation program that include both standard library support and floating-point math support.																
New CL options	<table> <tr> <th>Option</th><th>Action</th></tr> <tr> <td>/Oi</td><td>Generates intrinsic forms for certain library functions</td></tr> <tr> <td>/Ol</td><td>Enables loop optimizations</td></tr> <tr> <td>/Op</td><td>Forces consistent precision in the results of floating-point math operations</td></tr> <tr> <td>/qc</td><td>Specifies compilation with the Microsoft QuickC Compiler</td></tr> <tr> <td>/Sl</td><td>Specifies line width for source listings</td></tr> <tr> <td>/Sp</td><td>Specifies lines per page for source listings</td></tr> <tr> <td>/Ss</td><td>Specifies subtitles for source listings</td></tr> </table>	Option	Action	/Oi	Generates intrinsic forms for certain library functions	/Ol	Enables loop optimizations	/Op	Forces consistent precision in the results of floating-point math operations	/qc	Specifies compilation with the Microsoft QuickC Compiler	/Sl	Specifies line width for source listings	/Sp	Specifies lines per page for source listings	/Ss	Specifies subtitles for source listings
Option	Action																
/Oi	Generates intrinsic forms for certain library functions																
/Ol	Enables loop optimizations																
/Op	Forces consistent precision in the results of floating-point math operations																
/qc	Specifies compilation with the Microsoft QuickC Compiler																
/Sl	Specifies line width for source listings																
/Sp	Specifies lines per page for source listings																
/Ss	Specifies subtitles for source listings																

	/St	Specifies titles for source listings
	/Tc	Specifies C source files for files without .C extensions
	/Zp	Specifies structure packing on given byte boundaries
const keyword	Declares that a value will not change during program execution.	
New pragmas	Pragma	Action
	alloc_text	Names the code segment used to allocate specified functions
	function	Disables intrinsic-function generation for particular functions
	intrinsic	Specifies functions that will have intrinsic forms generated
	loop_opt	Controls program loop optimization on a local basis
	pack	Specifies byte boundaries for structure packing
	same_seg	Provides information about far data allocation that the compiler uses to perform optimizations
New /INFORMATION linker option	Displays information about the linking process. See Section 12.2.3 of the Microsoft CodeView and Utilities manual for more information.	
Language changes	The C language syntax and semantics have been modified in certain cases to correspond with recent updates to the Draft Proposed American National Standard—Programming Language C (hereinafter referred to as “the ANSI C standard”). See Appendix B, “Converting from Previous Versions of the Compiler,” and Appendix A of the <i>Microsoft C Language Reference</i> for more information.	

New library functions	All library functions defined in the the ANSI C standard are supported except the functions added for international-language support. Some existing functions have been modified and enhanced. In addition, a set of graphics functions has been added. See Appendix B, "Converting from Previous Versions of the Compiler," and the <i>Microsoft C Run-Time Library Reference</i> for more information.
-----------------------	--

1.4 Notational Conventions

The following notational conventions are used throughout this manual:

Example of Convention	Description of Convention									
Examples	<p>The typeface shown in the left column is used to simulate the appearance of information that would be printed on the screen or by the printer. For example, the following command line is printed in this special typeface:</p> <pre>CL /FoOUT.OBJ /DTRUE=1 FILE.C</pre> <p>When discussing this command line in text, items appearing on the command line, such as OUT.OBJ, also appear in the special typeface.</p>									
Language elements	<p>Bold type indicates elements of the C language that must appear in source programs as shown. Text that is normally shown in bold type includes operators, keywords, library functions, commands, options, and preprocessor directives. Examples are shown below:</p> <table><tr><td>+=</td><td>#if defined()</td><td>int</td></tr><tr><td>if</td><td>/Fa</td><td>fopen</td></tr><tr><td>main</td><td>sizeof</td><td></td></tr></table>	+=	#if defined()	int	if	/Fa	fopen	main	sizeof	
+=	#if defined()	int								
if	/Fa	fopen								
main	sizeof									
COMMANDS, FILES, REGISTERS, ENVIRONMENT, VARIABLES, and MACROS	<p>Bold capital letters are used for the names of executable files and files provided with the product, and for environment variables, symbolic constants, and macros. Commands typed at the MS-DOS level are also capitalized. These commands include built-in MS-DOS commands such as SET, as well as</p>									

program names such as **CL**, **LINK**, and **LIB**. You are not required to use capital letters when you actually enter these commands.

placeholders

Words in italics are placeholders that you must supply in command-line and option specifications and in the text for types of information. Consider the following option:

/H number

Note that *number* is italicized to indicate that it represents a general form for the */H* option. In an actual command, you would supply a particular number for the placeholder *number*.

Occasionally, italics are also used to emphasize particular words in the text.

Missing code

.
. .
.

Vertical ellipses are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, the ellipses between the statements indicate that intervening program lines occur but are not shown:

```
count = 0;
.
.
.
*pc++;
```

[[optional items]]

Double square brackets enclose optional fields in command-line and option specifications. Consider the following option specification:

/D identifier[[= *[[string]]*]]

The placeholder *identifier* indicates that you must supply an identifier when you use the */D* option. The outer square brackets indicate that you are not required to supply an equal sign (=) and a string following the identifier. The inner square brackets indicate that you are not required to enter a string following the equal sign, but if you do supply a string, you must also supply the equal sign.

Single square brackets are used in C-language array declarations and subscript expressions. For instance, a *[10]* is an example of brackets in a C subscript expression.

Repeating
elements...

Horizontal ellipses are used in syntax examples to indicate that more items having the same form may be entered. For example, several paths can be specified in the **PATH** command, as shown in the following syntax:

PATH[[=]*path*];*path*]

{*choice1*|*choice2*}

Braces and a vertical bar indicate that you have a choice between two or more items. Braces enclose the choices, and vertical bars separate the choices. You must choose one of the items unless all of the items are also enclosed in double square brackets.

For example, the **/W** (warning-level) compiler option has the following syntax:

/W {**0** | **1** | **2** | **3**}

You can use **/W1**, **/W2**, or **/W3** to display different levels of warning messages or **/W0** to suppress all warning messages.

"Defined terms"

Quotation marks set off terms defined in the text. For example, the term "far" appears in quotation marks the first time it is defined.

Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than ". For example, a C string used in an example would be shown in the following form:

"abc"

KEY+KEY

Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+C. Small capital letters are used for the names of keys (RETURN). Key sequences to be pressed simultaneously are indicated by the key names in small caps separated by a plus sign (CTRL+C).

1.5 Books about C

The manuals in this documentation package provide a complete programmer's reference for Microsoft C. They do not, however, teach you how to program in C. If you are new to C or to programming, you may want to familiarize yourself with the language by reading one or more of the following books:

Hancock, Les, and Morris Krieger. *The C Primer*. New York: McGraw-Hill Book Co., Inc., 1982.

Hansen, Augie. *Proficient C*. Bellevue, Washington: Microsoft Press, 1986.*

Harbison, Samuel P., and Greg L. Steele. *C: A Reference Manual*. Englewood Cliffs, New Jersey: Prentice-Hall Software Series, 1987.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Kochan, Stephen. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, Inc., 1983.

Plum, Thomas. *Learning to Program in C*. Cardiff, New Jersey: Plum Hall, Inc., 1983.

Schildt, Herbert. *C Made Easy*. Berkeley, California: Osborne McGraw Hill, 1985.

Schustack, Steve. *Variations in C*. Bellevue, Washington: Microsoft Press, 1985.

These books are listed for your convenience only. Except for its own publications, Microsoft Corporation does not endorse these books or recommend them over others on the same subject.

1.6 Requesting Assistance

If you feel you have discovered a problem in the software, please report the problem, using the Product Assistance Request at the back of this manual.

If you have comments or suggestions regarding any of the manuals accompanying this product, please use the Documentation Feedback Card at the back of this manual.

*Microsoft Press books are available wherever books and software are sold. To order by phone, call 1-800-638-3030; in Maryland, call collect 824-7300. For a complete catalog of Microsoft Press books, write to: Microsoft Press, 16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717.

CHAPTER

2

GETTING STARTED

2.1	Introduction	15
2.2	Backing Up Your Disks.....	15
2.3	Disk Contents.....	15
2.4	The SETUP Program.....	16
2.4.1	What SETUP Does	16
2.4.2	Running SETUP	17
2.4.2.1	Choosing SETUP Command-Line Options	18
2.4.2.2	Choosing Floating-Point Math Packages.....	20
2.4.2.3	Adding Graphics to Combined Libraries	21
2.4.2.4	Building Libraries	22
2.4.2.5	Compiling Programs after Installation ..	23
2.4.3	Library-Naming Conventions.....	23
2.4.4	How SETUP Organizes Files.....	24
2.4.5	Setting Up the Environment.....	26
2.4.6	Using Uncombined Libraries	26
2.4.7	Using the Microsoft C Graphics Library.....	28
2.5	Understanding the Compiler Software.....	28
2.5.1	Executable Files	28
2.5.2	Include Files	29
2.5.3	Library Files	30
2.5.4	Other Files.....	30
2.5.5	The Compiler Environment	32
2.5.6	Environment Variables.....	33
2.5.6.1	The PATH Variable	34
2.5.6.2	The LIB Variable	35

	2.5.6.3	The INCLUDE Variable	35
	2.5.6.4	The TMP Variable	35
	2.5.6.5	The CL Variable.....	35
	2.5.6.6	Defining Environment Variables.....	36
	2.5.6.7	Environment Variables and CL Options.....	37
	2.5.7	The CONFIG.SYS File	37
2.6		Using a RAM Disk	38
2.7		Converting Existing C Programs	38
2.8		Sample Compiler Command Lines	39
	2.8.1	Listing CL Options	39
	2.8.2	Simple Compile and Link	40
	2.8.3	Using Wild-Card Characters.....	40
	2.8.4	Compiling without Linking.....	40
	2.8.5	Using the Alternate Math Library	40
	2.8.6	Preparing to Use the CodeView Debugger.....	41
	2.8.7	Setting Titles and Subtitles.....	41
2.9		Practice Session	41

2.1 Introduction

This chapter explains how to use the **SETUP** program to install the compiler software on a hard disk and set up an operating environment for the compiler.

To get your C compiler up and running requires that you do the following:

1. Back up your disks (see Section 2.2).
2. Check the contents of the disks (see Section 2.3).
3. Read the **README.DOC** file on the Master distribution disk to learn about changes and additions made to the software after this manual was printed.
4. Run the **SETUP** program to install the software.
5. Read Section 2.8, "Sample Compiler Command Lines," or Section 2.9, "Practice Session," to learn how to compile and link.

Several MS-DOS procedures are mentioned in this chapter. In particular, the MS-DOS **SET** and **PATH** commands are used to give values to environment variables, which control the compiler environment. If you are unfamiliar with the **SET** and **PATH** commands, or with other MS-DOS procedures mentioned in this chapter, consult your DOS user's guide for instructions.

2.2 Backing Up Your Disks

After you have unwrapped your system disks, you should first make working copies, using the MS-DOS **COPY** command or the **DISKCOPY** utility. Save the original disks for making future working copies.

2.3 Disk Contents

When you first open your compiler package, you may want to verify that you have a complete set of software. One of the distribution disks in your compiler package contains a file named **PACKING.LST** on the Master distribution disk. This file lists and describes the files that make up the compiler software. It also lists the manuals and other materials included in the package that help you use the software.

You can use the **PACKING.LST** file to get a quick overview of the compiler software, as well as to verify that your software package is complete.

Note

Named disks included with the Microsoft C Optimizing Compiler are referred to as distribution disks to distinguish them from disks you create and label as you use the **SETUP** program.

2.4 The SETUP Program

The **SETUP** program is a set of MS-DOS batch files that automatically install the compiler software. You will find the **SETUP** program on Disk 1, the Master distribution disk (the disk may contain other files as well). The following sections explain what **SETUP** does and how to start **SETUP**.

2.4.1 What SETUP Does

The **SETUP** program performs the following tasks:

- Copies all necessary files to the directories or disks you specify.
- Builds run-time libraries based on your specifications. Each library includes support for the math, memory-model, and compatibility options you choose when you run **SETUP**. Under most circumstances, only one library is needed when you link.
- Creates a batch file named **NEW-VARS.BAT** that you can use to set the values of your environment variables so that the compiler and linker can find the files they need.
- Creates a file named **NEW-CONF.SYS** containing the appropriate settings for the **files** and **buffers** parameters in your **CONFIG.SYS** file.

See the **PACKING.LST** file on the Master distribution disk for a complete list of the files provided with the Microsoft C Optimizing Compiler. See Section 2.5.5, "The Compiler Environment," for more information about environment variables and the **CONFIG.SYS** file.

2.4.2 Running SETUP

To use the **SETUP** program, follow these two steps:

1. Invoke the **SETUP** program and specify the following information:
 - The memory model(s) you will use for your programs
 - Whether you will be linking with modules created by Versions 3.0 and 4.0 of Microsoft C
 - The directories where you want to install the various compiler files
2. At the **SETUP** prompt, type the names of the floating-point math packages you will use for your programs.

Based on the options you choose on the command line and the answers you give to the prompts, **SETUP** installs the compiler software and builds the appropriate libraries for the memory-model and floating-point options you have chosen.

Note

You may get an “out of environment space” message while running the **SETUP** program. If you get such a message, you need to expand the amount of environment space on your system. If you are using IBM PC-DOS Version 3.1 or earlier, you may be able to use the **SETENV** utility to expand your environment. Otherwise, you can specify a larger environment space in the **SHELL** command in your **CONFIG.SYS** file, then reboot your system. See the Microsoft Code-View and Utilities manual for a description of the **SETENV** utility; see the description of the **CONFIG.SYS** file in your DOS user’s guide for a description of the **SHELL** command.

To familiarize yourself with the options you will choose when you run **SETUP**, see Chapter 6, “Working with Memory Models,” for descriptions of memory models and their uses and Chapter 7, “Controlling Floating-Point Math Operations,” for descriptions of the available math packages and their uses.

2.4.2.1 Choosing SETUP Command-Line Options

Before you run **SETUP**, insert the Master distribution disk in a floppy-disk drive and make that the current drive.

To start **SETUP**, type a command line of this form and press ENTER:

SETUP *base models* [*cmpat*] [*bindir*] [*incldir*] [*libdir*] [*srcdir*]

Warning

The arguments you give **SETUP** include the names of several directories. If the directory you specify does not exist, **SETUP** creates the directory automatically.

Be careful not to give **SETUP** the name of an existing directory, unless you know that no files in that directory have the same names as the compiler files, since **SETUP** overwrites the existing files as it is installing.

To tell **SETUP** to use the appropriate default for an optional argument, simply omit the argument. Some optional arguments allow you to type a question mark (?) to choose the default.

The following list explains each of the arguments you give on the **SETUP** command line:

Argument	Meaning
<i>base</i>	The name of the "base" directory for the installation. This name must begin with a drive name (for example, C:). All other directories that you give on the command line are created as subdirectories of the base directory.
<i>models</i>	One or more letters, separated by spaces, telling SETUP which memory models you will use for your programs. Type S for small model, M for medium model, C for compact model, or L for large or huge model. SETUP uses the letters you type to determine which combined libraries to build. Because the combined libraries are large, you should specify only the memory models you know you will need for your programs. (If you use more than one memory model, you may want to

use the uncombined libraries instead of taking up disk space for combined libraries; see Section 2.4.6 for more information.)

cmpat

Type **c40** for this argument if you want compatibility with Version 4.0 of Microsoft C. If you ask for compatibility with Version 4.0, **SETUP** installs copies of the **SETARGV.OBJ** file under the names *mSETARGV.OBJ*, where *m* is one of the letters you typed for the *models* argument (**S** for small, **M** for medium, **C** for compact, and **L** for large and huge).

If you leave out this argument, **SETUP** installs only one copy, using the usual name, **SETARGV.OBJ**. See Section 2.5.4, "Other Files," for a description of **SETARGV.OBJ**.

bindir

The subdirectory of *base* where you want to install the compiler executable files, including the compiler, linker, and utilities. This argument must begin with a backslash (\).

If you leave out this argument or type a question mark (?) for it, **SETUP** uses the **\BIN** subdirectory by default.

includir

The subdirectory of *base* where you want to install include files. This argument must begin with a backslash (\).

If you leave out this argument or type a question mark (?) for it, **SETUP** uses the **\INCLUDE** subdirectory by default.

libdir

The subdirectory of *base* where you want to install library files. This argument must begin with a backslash (\).

If you leave out this argument or type a question mark (?) for it, **SETUP** uses the **\LIB** subdirectory by default.

srcdir

The subdirectory of *base* where **SETUP** copies the C start-up files (if desired) and where you will copy C source files. If specified, this argument must begin with a backslash (\).

If you type a question mark (?) for this argument, **SETUP** uses the **\SRC** subdirectory by default. If you do not give this argument, **SETUP** does not create a subdirectory for source files.

In addition to the subdirectories you tell **SETUP** to create, it creates the following subdirectories automatically:

- A subdirectory named **\TMP**, which the compiler will use for temporary files during compilation.
- A subdirectory named **\bindir\SAMPLE**, where **SETUP** installs the demonstration programs provided with the Microsoft C Optimizing Compiler.
- One or more subdirectories named **base\srcdir\model**, where *model* is one of the letters representing a memory model (**S** for small model, **M** for medium model, **C** for compact model, or **L** for large or huge model). Each **\srcdir\model** subdirectory contains one file named **VERSION.INC**. **SETUP** creates these subdirectories only if you have created a subdirectory for your source files.

■ Examples

```
SETUP S C:\ ? ? ?
```

The command line above tells **SETUP** to install the compiler software in the default subdirectories of the root directory (****). The default subdirectories are **\BIN** for compiler and utility executable files, **\INCLUDE** for include files, and **\LIB** for library files. No subdirectory is created for source files. Only the small-model library files are built.

```
SETUP C:\C S M C L C40 \BINDIR \INC \LIBS \SOURCES
```

The command line above tells **SETUP** to install the compiler software in the given subdirectories of the **\C** directory. Executable files are installed in the **\C\BINDIR** subdirectory; include files are installed in the **\C\INC** subdirectory; library files are installed in the **\C\LIBS** subdirectory; source files are installed in **\C\SOURCES** subdirectory; and demonstration files are installed in the **\C\BINDIR\SAMPLE** subdirectory. Library files are built, and **mSETARGV.OBJ** files are installed, for all available memory models.

2.4.2.2 Choosing Floating-Point Math Packages

After you enter the **SETUP** command line, **SETUP** displays a message telling you that it is ready to build combined run-time libraries. It then prompts you as shown below:

```
Do you wish to use the Emulator floating point
math package? [y/n]
```

Do you wish to use the 8087/80287 floating point math package? [y/n]

Do you wish to use the Alternate floating point math package? [y/n]

After each prompt:

- Type Y or y and press ENTER if you will use the given floating-point math package for your programs and you want to build combined libraries to support that package. Because the combined libraries are large, you should type **.L Y** only for the floating-point packages you know you will need for your programs. (If you use more than one floating-point math package, you may want to use the uncombined libraries instead of taking up disk space for combined libraries; see Section 2.4.6 for more information.)
- Type N or n and press ENTER if you will not use the given floating-point math package.

If you do not answer Y or y to at least one of the prompts, **SETUP** displays the following message:

You did not specify a floating point math option and, thus, setup will not build any combined libraries.
Is this what you want? [y/n]

To answer this prompt:

- Type N or n and press ENTER if you still want to build combined run-time libraries. **SETUP** returns to the earlier prompts for floating-point math packages.
- Type Y or y and press ENTER if you do not want **SETUP** to build combined libraries. In this case, you must install the uncombined libraries manually; see Section 2.4.6, "Using Uncombined Libraries," for more information.

2.4.2.3 Adding Graphics to Combined Libraries

The last prompt that **SETUP** displays asks if you want to include Microsoft C graphics functions in the combined libraries:

Do you want the graphics package included in your combined libraries [y/n]?

To answer this prompt:

- Type Y or y if you will be using Microsoft C graphics functions in your programs and you want to include these functions in the combined libraries that **SETUP** builds. Choosing this option adds approximately 50K to each combined library. However, you do not need to specify the graphics library **GRAPHICS.LIB** when you link programs that use graphics.
- Type N or n if you do not want to include graphics in the combined libraries. If you want to use graphics functions in your programs but do not want to include the graphics package in your combined libraries, see Section 2.4.7, "Using the Microsoft C Graphics Library."

2.4.2.4 Building Libraries

If you have answered Y or y to at least one of the math-package prompts, **SETUP** displays the names of the combined libraries it is building in response to your choices of memory models (on the command line) and math packages (from your answers to prompts). **SETUP** builds combined libraries in order to speed linking. When **SETUP** has finished building libraries, it displays the following message:

Library build complete.

Setup no longer needs the library sub-components and you do not normally need them to compile and link your C program. Do you want to delete them? [y/n]

If you use combinations of memory models and floating-point math packages other than the models and math packages you specified to **SETUP**, you may want to keep the uncombined libraries. However, if you will only be using the models and math packages supported by your combined libraries, you can delete the uncombined libraries. Enter Y or y to delete the uncombined libraries, or enter N or n if you want to keep the uncombined libraries.

After it finishes building libraries, **SETUP** finishes the rest of the installation process, including building the **NEW-VARS.BAT** and **NEW-CONF.SYS** files. When the installation process is complete, **SETUP** displays the following message:

Done!

2.4.2.5 Compiling Programs after Installation

After you have installed the compiler software, use the following procedure to compile your programs:

1. Set up your environment as described in Section 2.4.5. You can type
`NEW-VARS`
to change environment variables so that you can use the Microsoft C Optimizing Compiler immediately.
2. Use the MS-DOS **CD** command to move to the directory containing your program.
3. Type a **CL** command line to start compiling. (See Chapters 3, 6, and 7 for descriptions of the options that you can specify on the **CL** command line to control the compilation process.)

You can run the **SETUP** program without reading any further in this section, since **SETUP** provides all the information you need. However, you may find the information in the following sections helpful if you should run into problems.

2.4.3 Library-Naming Conventions

SETUP gives the libraries it builds default names based on the memory models and math packages you choose. Each default name has the following form:

{S | M | C | L}LIBC{E | 7 | A}.LIB

The first character of the library base name is determined by the memory model you choose: **S** if you choose the small (default) memory model, **M** if you choose the medium memory model, **C** if you choose the compact memory model, or **L** if you choose the large or huge memory model.

The last character of the default library base name is determined by the math package you choose: **E** if you choose the emulator package, **7** if you choose the 8087/80287 math package, or **A** if you choose the alternate math package.

If you change the library name that **SETUP** assigns, you must explicitly specify the new library name when you link your program. (If you do not specify the new name, the linker expects that the library name is the default for the floating-point and memory-model compiler options you choose.)

Note

For ease of discussion, the remainder of this manual uses the default names to identify libraries that support particular combinations of memory models and math packages.

2.4.4 How SETUP Organizes Files

The following lists show each subdirectory of the base directory that **SETUP** uses by default and the files that it copies to each default subdirectory. Remember that you can tell **SETUP** to use subdirectory names other than the defaults; the same files are copied to the directory you specify in this case.

\BIN subdirectory:

C1.ERR	CL.EXE	EXEPACK.EXE
C1.EXE	CL.HLP	LIB.EXE
C2.EXE	CV.EXE	LINK.EXE
C23.ERR	CV.HLP	MAKE.EXE
C3.EXE	ERROUT.EXE	SETENV.EXE
CL.ERR	EXEMOD.EXE	

\INCLUDE subdirectory:

ASSERT.H	FLOAT.H	SEARCH.H	STDLIB.H
CONIO.H	IO.H	SETJMP.H	STRING.H
CTYPE.H	LIMITS.H	SHARE.H	TIME.H
DIRECT.H	MALLOC.H	SIGNAL.H	VARARGS.H
DOS.H	MATH.H	STDARG.H	
ERRNO.H	MEMORY.H	STDDEF.H	
FCNTL.H	PROCESS.H	STDIO.H	

\INCLUDE\SYS subdirectory:

LOCKING.H
STAT.H
TIMEB.H
TYPES.H
UTIME.H

\LIB subdirectory:

mLIBCf.LIB

Note that the **LIB** environment variable is not used to find the **mVARSTCK.OBJ**, **SETARGV.OBJ**, and **BINMODE.OBJ** files; if these files are not in your current working directory, you must specify a path name at link time.

Note

Throughout the remainder of this manual, the convention **mLIBCf.LIB** is used to refer to the standard libraries built by **SETUP**. In this convention, the *m* refers to the standard memory model that the library supports: **S** for small model (the default), **M** for medium model, **C** for compact model, or **L** for large or huge model. The *f* refers to the floating-point math package that the library supports: **E** for the emulator package, **7** for the 8087/80287 package, and **A** for the alternate math package.

This convention is also used for other files, such as **mVARSTCK.OBJ**, that are supplied in multiple copies to handle each standard memory model.

\BIN\SAMPLE subdirectory:

CIRCLE.C	DEMO.C	S1.@ @ @
CIRCLE.EXE	MATH.C	S2.@ @ @
CIRCLE.R	MENU.BAT	S3.@ @ @
CIRCLEB.BAT	NEW-CONF.SYS	S4.@ @ @
COUNT.C	NEW-VARS.BAT	S5.@ @ @
COUNT.EXE	PI.C	S6.@ @ @
COUNT.R	PI.EXE	S7.@ @ @
COUNT.TXT	PIB.BAT	S8.@ @ @
COUNTB.BAT	RESPOND.COM	S9.@ @ @
DEMO.BAT	SAMPLE.BAT	

\SRC subdirectory:

BRKCTL.INC	EXECMSG.ASM	STDALLOC.ASM
CHKSTK.ASM	FMSGHDR.ASM	STDARGV.ASM
CHKSUM.ASM	HTOI.C	STDENV.P.ASM
CMACROS.INC	HTOI.EXE	WILD.C
CRT0.ASM	HTOI.OBJ	
CRT0DAT.ASM	MSDOS.H	
CRT0FP.ASM	MSDOS.INC	
CRT0MSG.ASM	NMSGHDR.ASM	
DOSSEG.C	REGISTER.H	
EMOEM.ASM	SETARGV.ASM	

2.4.5 Setting Up the Environment

SETUP automatically creates a batch file named **NEW-VARS.BAT** in the **\BIN\SAMPLE** subdirectory of your base directory. You can use **NEW-VARS.BAT** to change the values of your environment variables so that the compiler and linker can find the files they need. If you choose to install the compiler files in the default subdirectories, the **NEW-VARS.BAT** program sets these variables as shown in Table 2.1.

Table 2.1
Default Environment Settings

Variable	Path
PATH	<i>base\</i> BIN
INCLUDE	<i>base\</i> INCLUDE
LIB	<i>base\</i> LIB
TMP	<i>base\</i> TMP

Ordinarily, no temporary files will remain in the **\TMP** subdirectory, since the **CL.EXE** program automatically removes them by the time processing finishes. However, if you abort a compilation, you may find temporary files remaining in the **\TMP** subdirectory.

If you wish, you can add the **SET** commands in the **NEW-VARS.BAT** file to your **AUTOEXEC.BAT** file so that the environment is set up correctly each time you reboot.

In addition to **NEW-VARS.BAT**, **SETUP** creates a file named **NEW-CONF.SYS** in the **\BIN\SAMPLE** subdirectory. This file sets the **files** and **buffers** parameters to appropriate values for the Microsoft C Optimizing Compiler. You can either replace your existing **CONFIG.SYS** file with **NEW-CONF.SYS** or copy the **buffers** and **files** settings from **NEW-CONF.SYS** to your existing **CONFIG.SYS** file.

2.4.6 Using Uncombined Libraries

The **SETUP** program builds combined libraries because linking with combined libraries is faster than linking with uncombined libraries. However, if you use many different combinations of memory models and floating-point math packages, you may not want to use up the disk space required for all of the combined libraries you need.

If you choose not to combine libraries, you can copy the appropriate uncombined libraries to the subdirectory you chose for libraries (by default, *base\LIB*). The following uncombined libraries are provided with the Microsoft C Optimizing Compiler (with *m* indicating the appropriate memory model):

Library	Purpose
<i>m</i>LIBC.LIB	Standard run-time library; contains all of the routines included in the Microsoft C run-time library except math routines that require floating-point support.
<i>m</i>LIBFP.LIB	Floating-point math library; required whenever your program uses EM.LIB or 87.LIB .
<i>m</i>LIBFA.LIB	Alternate floating-point library; can be used instead of EM.LIB and <i>m</i>LIBFP.LIB when speed is more important than precision in floating-point calculations. See the discussion of floating-point options in Chapter 7, "Controlling Floating-Point Math Operations," for more information.
EM.LIB	Model-independent floating-point emulator; used to perform floating-point operations.
LIBH.LIB	Model-independent "compiler helper" functions; used to handle complex operations such as 32-bit multiplication and division.
87.LIB	Model-independent 8087/80287 floating-point library; provides minimal floating-point support and can only be used when an 8087 or 80287 coprocessor is present.

The following list shows each combined library built by **SETUP** and the corresponding uncombined libraries:

Combined Library	Uncombined Libraries
<i>m</i>LIBCE.LIB	<i>m</i>LIBC.LIB , <i>m</i>LIBFP.LIB , LIBH.LIB , and EM.LIB
<i>m</i>LIBC7.LIB	<i>m</i>LIBC.LIB , <i>m</i>LIBFP.LIB , LIBH.LIB , and 87.LIB
<i>m</i>LIBCA.LIB	<i>m</i>LIBC.LIB , <i>m</i>LIBFA.LIB , and LIBH.LIB

When you compile and link a program, you must give the **/NOD** linker option after the **/link** option on the **CL** command line and specify the uncombined libraries for the memory-model option and floating-point option you are using. See Section 4.3.1 for more information about specifying uncombined libraries.

2.4.7 Using the Microsoft C Graphics Library

If you decided not to include graphics in the combined libraries built by **SETUP**, but you still want to use Microsoft C graphics routines in your programs, you must explicitly link with the **GRAPHICS.LIB** library in addition to the appropriate combined library (or uncombined libraries). First, copy **GRAPHICS.LIB** to the subdirectory where you installed the other libraries. You then have the following alternatives:

- Give **GRAPHICS.LIB** explicitly on the **CL** command line. See Sections 4.3.1 and 4.3.1.1 for more information about specifying additional libraries on the **CL** command line.
- Specify **GRAPHICS.LIB** in the **CL** environment variable. This tells the **CL** command to link with **GRAPHICS.LIB** automatically. See Section 3.2.2 for information about the **CL** environment variable.

2.5 Understanding the Compiler Software

Sections 2.5.1 through 2.5.7 provide background information about the Microsoft C Optimizing Compiler software and the environment in which it operates. This information is not required to use the compiler; however, it may help you better understand the individual components of the compiler software and how they work together.

Section 2.5.1, "Executable Files," Section 2.5.2, "Include Files," and Section 2.5.3, "Library Files," describe the three main categories of files that make up the Microsoft C Optimizing Compiler. Section 2.5.4 describes several additional files that do not fall into the three main categories.

Sections 2.5.5 through 2.5.7 describe the compiler environment and the ways in which you can control this environment.

2.5.1 Executable Files

Executable files have an **.EXE** extension. The following executable files are provided with the Microsoft C Optimizing Compiler:

File	Purpose
CL.EXE	Control program for the compiler and linker.
C1.EXE, C2.EXE, C3.EXE	The three compiler stages, or “passes,” which are executed in order when you process a file using CL.EXE .
LINK.EXE	Linker utility, which produces an executable program file from your compiled files. LINK.EXE can either be automatically invoked by CL.EXE or invoked separately.
CV.EXE	The Microsoft CodeView symbolic debugger.
LIB.EXE	Library-manager program that creates and organizes libraries of object modules.
EXEPACK.EXE	Utility used to pack executable files.
EXEMOD.EXE	Utility that changes the headers of executable files.
SETENV.EXE	Utility that changes the size of the MS-DOS environment table.
ERROUT.EXE	Utility that redirects standard-error output.

See Chapter 3 of this manual for information about the **CL** command. See the Microsoft CodeView and Utilities manual for information about the **LINK**, **CV**, **LIB**, **MAKE**, **EXEPACK**, **EXEMOD**, **SETENV**, and **ERROUT** utilities.

2.5.2 Include Files

Include files have an extension of **.H**. Include files are C source files you can incorporate into your program by using the C preprocessor directive **#include**. These files contain definitions used by run-time library routines.

By convention, some include files are stored in a subdirectory named **\SYS**. This convention originated with the practice of storing files that define “system-level” constants and types in a separate “system” subdirectory on UNIX® and XENIX systems. However, not all the include files that are traditionally stored in the **\SYS** subdirectory contain system-level definitions, and some of the include files *not* in the **\SYS** subdirectory contain system-level definitions. Since many programs, particularly those created under the XENIX and UNIX operating systems, rely on the **\SYS** subdirectory convention, Microsoft continues to recognize this convention in order to to maintain compatibility with existing programs.

2.5.3 Library Files

Library files contain compiled run-time library routines to be linked with your program. The **SETUP** program builds a library for each combination of memory models and math packages you specify. In most cases, this library is the only library you need to use for linking programs. See Section 2.4.3 for information about library-naming conventions.

Each library built by **SETUP** also contains an object module named **CRT0.OBJ**, which is the program start-up routine. This routine performs the following important tasks:

- Allocates the stack for your program and initializes the segment registers
- Sets up the *argv*, *argc*, and *envp* variables to allow command-line arguments and environment settings to be passed to the program
- Sets up and maintains the operating environment for the program
- Initializes the emulator, if the program uses the emulator

2.5.4 Other Files

Several of the files provided in the Microsoft C Optimizing Compiler package don't fit into any of the categories discussed so far. The following list describes these files:

File	Purpose
*.HLP	Help files for the CL compiler driver (CL.HLP) and the Microsoft CodeView symbolic debugger (CV.HLP).
*.ERR	Error-message files for the CL compiler driver (CL.ERR) and compiler passes (C1.ERR and C23.ERR).
<i>m</i> VARSTCK.OBJ	Object files that, when linked with a program, allow the heap to compete with the stack for memory space. In this way, the heap can allocate memory from unused stack space. See Section 3.3.21, "Controlling Stack and Heap Allocation," for more information about the <i>m</i> VARSTCK.OBJ files.

BINMODE.OBJ	Object file used to change the default mode for data files from text mode to binary mode. This file can be used with all five memory models. See Section 3.4 of this manual, "Controlling Binary and Text Modes," for information about using BINMODE.OBJ .
COUNT.*	Files used in the practice session for the Microsoft CodeView symbolic debugger.
DEMO.C	C program used in the sample compile-and-link session described in Section 2.9, "Practice Session." Other demonstration programs may be included on your distribution disks. If so, they are described in the README.DOC file.
EMOEM.ASM	Assembly-language program that allows you to customize floating-point software. See Section 7.7, "If Your Computer Is Not IBM Compatible," for more information about EMOEM.ASM .
SETARGV.OBJ	<p>Object file containing a routine that expands the MS-DOS wild-card characters (?) and (*) in file-name arguments passed to C programs from the command line. Wild-card expansion is performed only if you explicitly link with SETARGV.OBJ. This file can be used with all five memory models. For more information about SETARGV.OBJ, see Section 5.2, "Passing Command-Line Data to a Program."</p> <p>Note that if you have chosen compatibility with Version 4.0 of Microsoft C, SETUP installs versions of SETARGV.OBJ under the name <i>mSETARGV.OBJ</i>, where <i>m</i> specifies the memory model. If you link with object files compiled with Version 4.0 of Microsoft C, you must link with the version that is appropriate for the memory model you are using. (See Chapter 6, "Working with Memory Models," for more information about memory models.)</p>

Start-up source files

Source files for the Microsoft C Optimizing Compiler start-up module are provided for users who wish to modify them for purposes of developing read-only memory programs or memory-resident programs. See the file named **README.DOC** in your *base\SRC* sub-directory (or the directory you chose for source files) for more information about these files.

Note

Except for these source files, no support is provided for modification of the start-up code. The start-up source files are subject to change in future versions of the Microsoft C Optimizing Compiler.

2.5.5 The Compiler Environment

The compiler environment consists of environment variables that tell the compiler and linker where to find the files they need to process a program. They are called "environment" variables because they define the environment in which the compiler and linker operate. Environment variables are defined at the MS-DOS command level using the MS-DOS commands **SET** and **PATH**.

The **SETUP** program creates a batch file named **NEW-VARS.BAT** that automatically sets the values of environment variables. You can either type

NEW-VARS

to set up the environment for the Microsoft C Optimizing Compiler or copy the settings in **NEW-VARS.BAT** to your **AUTOEXEC.BAT** file so that the environment is set up correctly each time you reboot.

NEW-VARS.BAT does not set the value of one environment variable that is useful during compiling and linking: **CL**. This variable allows you to specify default options and input files for the **CL** command.

In addition to changing the values of your environment variables, you may have to change your **CONFIG.SYS** file so that it satisfies the compiler requirements. The **SETUP** program also creates **NEW-CONF.SYS**, a

file that contains settings that you can copy to your **CONFIG.SYS** file so that the compiler can work correctly.

The following sections describe the environment variables and **CONFIG.SYS** settings used by the compiler and linker.

2.5.6 Environment Variables

The **CL.EXE** compiler driver looks for four environment variables: **PATH**, **INCLUDE**, **TMP**, and **CL**. The linker, which is invoked by **CL.EXE**, looks for two environment variables: **LIB** and **LINK**. (See the Microsoft CodeView and Utilities manual for a description of the **LINK** environment variable.)

The **PATH**, **INCLUDE**, **TMP**, and **LIB** variables are assigned one or more path specifications (in the case of **TMP**, only one path specification) that tell the compiler or linker where to find a particular type of file, as shown in the following list:

Environment Variable	Type of File
PATH	Executable files (any file ending with .EXE). These include the compiler control program (CL.EXE), the compiler passes, the linker, and all of the utilities.
LIB	Library files (any file ending with .LIB).
TMP	Temporary files created by the compiler; only one path specification may be used.

Note

If you have a memory-based disk emulator, commonly referred to as a "RAM disk," you can compile programs faster by assigning the drive name for the RAM disk to the **TMP** variable.

INCLUDE	Include files.
----------------	----------------

The compiler or linker always searches the current working directory first before searching the locations given in an environment variable. Exceptions to this sequence are **#include** files, which are enclosed in angle brackets (`< >`).

The **CL** variable does not tell the compiler and linker where to find files; instead, it defines default options for the **CL** command. It can also specify source files, object files, or libraries, although this use is less common.

Although environment variables are usually helpful, you are not required to set them. If you do not set these variables, the current working directory is used to search for files and create temporary files.

2.5.6.1 The PATH Variable

The **CL** command searches for the compiler and linker in the following order:

1. In the directory where **CL.EXE** resides (which, under Version 2.x of MS-DOS, always looks like the current working directory)

Note

This manual refers to all versions of a product in, for example, the Version 2 range, as Version 2.x.

2. In the current working directory
3. In all directories specified in the **PATH** command, in order of their appearance

This search order makes it easy to store and use multiple versions of the compiler without worrying about using the wrong version.

MS-DOS also uses the **PATH** setting to locate executable files. For example, when you invoke **CL.EXE** (by typing **CL**), the MS-DOS system finds **CL.EXE** by looking in your default directory and in the directories specified in the **PATH** setting. If you include the path name of the directory containing **CL.EXE** in your **PATH** setting, you can execute the control program from any directory.

2.5.6.2 The LIB Variable

The **LIB** environment variable defines where the linker searches for libraries. (Section 4.3.1 gives the rules the linker follows when searching for libraries.) This variable can contain one or more path specifications, separated by semicolons.

When you compile a source file using the Microsoft C Optimizing Compiler, the compiler places a library name in the object file it creates. This is the name of the library that supports the memory-model and floating-point options you have given on the **CL** command line.

The linker searches the standard places for this library. The linker also uses the **LIB** setting to search for any other libraries that you specify on the command line at link time. See Section 4.3.1.3 for more information about changing libraries at link time.

2.5.6.3 The INCLUDE Variable

The **INCLUDE** environment variable defines the standard places where the compiler searches for each include file (a file incorporated into another source file with the **#include** preprocessor directive). The **/I** and **/X** options, discussed in Section 3.3.9.6, let you temporarily change the search path for include files without affecting the **INCLUDE** variable. Section 3.3.9.6 also lists the places that the compiler searches for include files and the order in which these places are searched.

2.5.6.4 The TMP Variable

The compiler creates a number of temporary files as it processes a program. The **TMP** environment variable tells the compiler and the operating system where to create these files. The temporary files are removed by the time the compiler finishes processing.

The space required for the temporary files is typically double the size of the source file. It is often helpful to create the temporary files on a memory-based disk emulator, commonly referred to as a “RAM disk,” as described in Section 2.6. You can speed processing by assigning the drive name you use for a RAM disk to the **TMP** variable.

2.5.6.5 The CL Variable

The **CL** environment variable allows you to define default options for the **CL** compiler driver. This variable is useful if you usually give a large number of options or if you usually use the same set of options when you

compile and link. Since the options you define in the environment are not counted in the 128-character limit for the command line, you can define the options you use most often with the **CL** variable and then give only the options you need for specific purposes on the command line.

The **CL** variable may also name source files, object files, and libraries. Any specified source files are compiled; if you also link with the **CL** command, the specified object files are linked and the specified libraries are searched.

The options and files in the **CL** variable are treated just as if you typed them on the command line following **CL** and before the rest of the command line. Conflicts between options given in the environment and options given on the command line are handled accordingly.

Note that, in most cases, if you define a **CL** or linker option in the environment, you cannot turn the option off from the command line. In cases where you do not want to use an option, you must reset the **CL** environment variable and omit the option that you do not want to use. (See Section 3.3, "Using CL Options," for exceptions to this rule.)

2.5.6.6 Defining Environment Variables

Use the MS-DOS **SET** command to define the values of **INCLUDE**, **LIB**, **TMP**, and **CL**. Use the MS-DOS **PATH** command to define the value of **PATH**.

You must set the values of **PATH**, **INCLUDE**, and **TMP** *before* invoking the compiler if you want the variables to be effective while the compiler is running. Similarly, you must set **LIB** before the linking stage.

The **SET** command has the following format for the **INCLUDE**, **LIB**, and **TMP** variables:

```
SET variable= path[:path]...
```

The **TMP** variable can be assigned only one path name. The **INCLUDE** and **LIB** variables can each contain more than one path name. See Section 3.2.2 for information about setting the **CL** variable.

The **PATH** command has the following format:

```
PATH[= ] path[:path]...
```

For example, you might use the following command line:

```
PATH C:\BIN;C:\LINKER
```

This tells the compiler and the operating system to search for executable files on Drive C in the directory named \BIN, then, if necessary, in the \LINKER directory. Although you are allowed to define the **PATH** variable with the **SET** command, using this method under versions of MS-DOS earlier than 3.0 can cause the **PATH** variable to work incorrectly for some directory specifications using lowercase letters.

Note

The environment table, which holds any environment variables you have set and the values you have assigned, is 160 bytes by default. If you want to set up a complex environment, this may not be enough space. If you are running on IBM PC-DOS Version 3.1 or earlier, you can use the **SETENV** program to increase the size of the environment table. See Section 15.3 of the Microsoft CodeView and Utilities manual for more information.

Once you have set an environment variable, it remains in effect until you reset it to a different value (or to an empty value) or until you turn off your machine.

2.5.6.7 Environment Variables and CL Options

Certain command-line options available with the compiler override the effect of environment variables. For example, the **/X** option (described in Section 3.3.9.6) tells the compiler not to automatically search the standard places for include files. The result is that the compiler does not search for include files in the directories specified by the **INCLUDE** variable.

2.5.7 The CONFIG.SYS File

Before you can run the compiler you must make sure that your **CONFIG.SYS** file allows the compiler to open 20 files. The **NEW-CONF.SYS** file created by **SETUP** contains the following line:

```
files=20
```

If the **files=** line in your **CONFIG.SYS** file specifies a number less than 20, replace the line in your **CONFIG.SYS** file with the line from the **NEW-CONF.SYS** file. If you do not currently have a **CONFIG.SYS** file, copy the **NEW-CONF.SYS** file to a **CONFIG.SYS** file in the root directory, then reboot, before you compile programs.

Note

If you do not specify at least 15 files in the **CONFIG.SYS** file, you may see one of the following fatal error messages during compilation:

```
fatal error C1041: Cannot open compiler intermediate file
- no more files
```

or

```
fatal error C1015: Cannot find 'includefile'
```

It is recommended, though not required, that you also set the number of buffers allowed in your **CONFIG.SYS** file. Check your **CONFIG.SYS** for the following line:

```
buffers=number
```

If *number* is not already set, 10 is a reasonable number to choose.

2.6 Using a RAM Disk

If your computer has sufficient available memory, you can set it up to run portions of the compiler from a memory-based disk emulator, also known as a RAM disk. Using a RAM disk allows you to compile programs considerably faster than you could otherwise.

If you are using a RAM disk, you can set the value of the **TMP** environment variable to the drive name you are using for the RAM disk. In this way, you can use the RAM disk for temporary files during compilation. Since temporary files are typically twice the size of the source file, you need approximately twice as much available memory as the size of the source file you are compiling.

2.7 Converting Existing C Programs

If you are using an earlier version of the Microsoft C Optimizing Compiler, or if you have programs written for such a compiler, turn to Appendix B for a discussion of differences between this compiler and earlier versions.

You may need to make minor changes to existing source programs to compile them with Version 5.0; however, recompiling these programs will generally result in improved performance.

2.8 Sample Compiler Command Lines

This section helps you quickly begin compiling and linking programs by giving examples of common **CL** command lines and options. For a step-by-step approach to the compiling and linking process, see Section 2.9, "Practice Session."

The command lines given in the following sections illustrate some of the most common command-line options. You can use these command lines exactly as shown to get started with the compiler and linker, or you can use them as models and supply your own combination of options.

See Chapter 3 for an in-depth discussion of how the **CL** command line works. Chapter 4 explains how to control linking using the **CL** command line. Chapter 12 of the Microsoft CodeView and Utilities manual fully describes the linker and its options.

Each option illustrated in this section is fully described elsewhere in this manual. Use the index at the back of this manual to find more information about particular options.

The **CL** command invokes the compiler, the linker, or both, so you do not need to give separate commands for compiling and linking (although you may). Notice that no library names are given at link time in the commands shown below, since you are not required to give a library name when you link unless you have changed the names of the libraries created by **SETUP**.

2.8.1 Listing CL Options

For a quick overview of commonly used compiler options, type the following at the MS-DOS prompt:

```
CL /HELP
```

The **/HELP** option displays a categorized summary of **CL** options.

The *Microsoft C Quick Reference Guide* that accompanies this manual is another good source for a quick overview. It lists the **CL** options in alphabetical order.

2.8.2 Simple Compile and Link

```
CL FILE1.C FILE2.C
```

The example above demonstrates compiling and linking two files named `FILE1.C` and `FILE2.C`. Two object files, `FILE1.OBJ` and `FILE2.OBJ`, are created. Since no memory-model or floating-point options are given, these object files are linked with the appropriate library for the default memory model (small) and floating-point math package (emulator): **SLIBCE.LIB**. The executable file is named `FILE1.EXE`.

2.8.3 Using Wild-Card Characters

```
CL /FePROGRAM /Fs *.C
```

The command above compiles and links all C source files in the current working directory. The `/Fe` option gives the resulting executable file the name `PROGRAM.EXE`. The `/Fs` option creates a source-listing file for each source file; each source-listing file has the same base name as the corresponding source file, but has the extension **.LST** instead of **.C**. (The base name of a file is the portion of the name preceding the period.)

2.8.4 Compiling without Linking

```
CL /c FILE.C
```

The command above compiles but does not link the given file. You can also use the **CL** command to link without compiling by just giving object files on the command line. For example,

```
CL FILE.OBJ
```

invokes the linker to create an executable program named `FILE.EXE`.

2.8.5 Using the Alternate Math Library

```
CL /FPa EMULAT.C
```

By default, Microsoft C programs handle floating-point operations by generating in-line instructions for an 8087 or 80287 math coprocessor; if a coprocessor is present, the program uses it, but if a coprocessor is not present, the program uses an emulator library instead.

The command shown above creates a program that handles floating-point math differently: the program generates calls to floating-point functions in an alternate math library (**SLIBFA.LIB**). The alternate math library provides the smallest, fastest option if no coprocessor is installed, although the program sacrifices some accuracy for speed. If a coprocessor is installed, the program ignores it. See Section 7.2.3 for information about this option.

2.8.6 Preparing to Use the CodeView Debugger

```
CL /Zi FILE.C
```

The example above uses the **/Zi** option to create object and executable files that contain symbol-table information for debugging with the Microsoft CodeView window-oriented debugger. When the **/Zi** option is given with no explicit optimization option, in order to make program debugging easier some complex optimizations are not performed. If you do not want the compiler to perform any optimization, specify the **/Od** option along with the **/Zi** option. See Section 3.3.12, “Preparing for Debugging,” for more information about the **/Zi** option and Section 3.3.13.1, “Controlling Optimization,” for more information about the **/Od** option.

2.8.7 Setting Titles and Subtitles

```
CL /Fs /St "Main Title" /Ss "Subtitle" /Sp20 /S190 FILE.C
```

The example above compiles and links **FILE.C**, creating an executable file named **FILE.EXE**. The **/Fs** option creates a source-listing file named **FILE.LST**. The listing has a main title and subtitle; it is 20 lines long and 90 characters wide. See Sections 3.3.8.1, “Types of Listings,” and 3.3.8.4, “Setting Titles and Subtitles,” for more information.

2.9 Practice Session

This section shows you the steps involved in compiling and linking a program using the Microsoft C Optimizing Compiler. By following these steps you can produce and run an executable program file.

The source file used for this practice session is the sample source file **DEMO.C**, which the **SETUP** program installs in your *base***SAMPLE** subdirectory. **DEMO.C** is a simple C program that contains only one function, the **main** function. The **main** function in this program displays any command-line arguments you pass to the program at execution time

and displays the current values of environment settings. See Chapter 5, "Running C Programs on MS-DOS," for a full discussion of passing command-line data to programs, accessing the program environment from within a program, and declaring the *argc*, *argv*, and *envp* parameters.

This practice session assumes that you have used the **SETUP** program to install the software and build the libraries you need; that you have set up the compiler environment using the **NEW-VARS.BAT** program created by **SETUP**; and that you have copied **DEMO.C** to the directory or disk where you want to do your compiling and linking.

You can verify that the compiler environment is set up correctly by typing **SET** and pressing ENTER. This command lists all environment variables and their current settings. Make sure the **PATH**, **INCLUDE**, **TMP**, and **LIB** variables are in the list and that they are set appropriately for your system. If you have installed the compiler using the **SETUP** program, these variables should have the values shown in the following list:

Variable	Path
PATH	<i>base\BIN</i>
INCLUDE	<i>base\INCLUDE</i>
LIB	<i>base\LIB</i>
TMP	<i>base\TMP</i>

If your settings do not match the above settings, turn back to Section 2.5.5 or 2.5.6 to review the disk setup and environment settings that are appropriate to your system.

Once you have set up the environment, you are ready to begin processing **DEMO.C** by using the following procedure:

1. Make sure that the directory containing **DEMO.C** is your current working directory. (Use the MS-DOS **CD** command to change directories, if necessary.)

2. Type

```
CL /Fs DEMO.C
```

First, the **CL** command invokes the compiler, which prints the following message on your screen and begins to compile the source file:

```
Microsoft (R) C Optimizing Compiler Version 5.00  
(C) Copyright Microsoft Corp 1984, 1985, 1986, 1987. All rights reserved.
```

The **/Fs** option creates a source listing named **DEMO.LST** in the current working directory.

3. The next message you see is similar to the following:

```
Microsoft (R) Overlay Linker Version 3.60  
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.
```

This means that compilation is completed and the file is now being linked to form an executable program.

4. When the linking process is finished, the MS-DOS prompt reappears. Your current working directory now has an executable file named **DEMO.EXE**. It also contains an object file named **DEMO.OBJ** and a source-listing file named **DEMO.LST**.

You may want to examine the source-listing file to familiarize yourself with its format. However, the file is not required for running the program, and you can delete it.

You can also delete the object file (**DEMO.OBJ**); since you have the executable program file, it is no longer needed.

5. You can run the sample program by simply typing **DEMO**. However, since the sample program is designed to take command-line arguments and print them, give command-line arguments when you run the program. For instance, you can run the program and pass three arguments by typing:

```
DEMO ONE TWO THREE
```

The program name is displayed on your screen, followed by the arguments **ONE**, **TWO**, and **THREE** and a listing of all current environment settings. The environment settings include **PATH**, **LIB**, **INCLUDE**, and **TMP**, as well as any other settings that are currently in effect (whether or not they apply to the C program or to the compilation and linking processes).

Note

Under versions of MS-DOS earlier than 3.0, the program name is not available. The letter **C** is always given as the program name.

CHAPTER

3

COMPILING WITH THE CL COMMAND

3.1	Introduction	47
3.2	The Basics: Compiling and Linking C Programs.....	48
3.2.1	The CL Command	48
3.2.1.1	Specifying Source and Object Files	48
3.2.1.2	Creating Executable Files	50
3.2.1.3	Stopping CL	51
3.2.2	Using the CL Environment Variable	51
3.3	Using CL Options	53
3.3.1	Memory-Model (/A) and Floating-Point (/FP) Options	54
3.3.2	Listing the Compiler Options (/HELP)	56
3.3.3	Specifying Source Files (/Tc).....	56
3.3.4	Compiling without Linking (/c)	57
3.3.5	Compiling with QuickC (/qc, /Zr, /Zq)	57
3.3.6	Naming the Object File (/Fo).....	58
3.3.7	Naming the Executable File (/Fe)	60
3.3.8	Creating Listing Files.....	61
3.3.8.1	Types of Listings (/Fs, /Fl, /Fa, /Fc, and /Fm)	61
3.3.8.2	Special File Names.....	64
3.3.8.3	Setting Line Width (/Sl) and Page Length (/Sp).....	65
3.3.8.4	Setting Titles (/St) and Subtitles (/Ss) ..	66
3.3.8.5	Formats for Listings.....	67
3.3.9	Controlling the Preprocessor.....	75
3.3.9.1	Defining Constants and Macros (/D)	75
3.3.9.2	Predefined Identifiers.....	77

3.3.9.3	Removing Definitions of Predefined Identifiers (/U, /u)	78
3.3.9.4	Producing a Preprocessed Listing (/P, /E, /EP)	79
3.3.9.5	Preserving Comments (/C)	80
3.3.9.6	Searching for Include Files (/I, /X)	80
3.3.10	Using the 80186, 80188, or 80286 Processor (/G0, /G1, /G2)	81
3.3.11	Checking for Program Errors	82
3.3.11.1	Understanding Error Messages	82
3.3.11.2	Setting the Warning Level (/W, /w)	84
3.3.11.3	Checking Syntax (/Zs)	85
3.3.11.4	Generating Function Declarations (/Zg)	86
3.3.12	Preparing for Debugging (/Zi, /Zd, /Od)	87
3.3.13	Optimizing	88
3.3.13.1	Controlling Optimization (/O Options)	89
3.3.13.2	Removing Stack Probes (/Gs)	97
3.3.14	Enabling and Disabling Language Extensions (/Ze, /Za)	99
3.3.15	Packing Structure Members (/Zp)	100
3.3.16	Setting the Stack Size (/F)	102
3.3.17	Restricting the Length of External Names (/H)	103
3.3.18	Labeling the Object File (/V)	103
3.3.19	Suppressing Default-Library Selection (/Zl)	104
3.3.20	Changing the Default char Type (/J)	105
3.3.21	Controlling Stack and Heap Allocation	105
3.3.22	Controlling the Calling Convention (/Gc)	106
3.3.23	Compiling for Windows Applications (/Aw, /Gw)	109
3.3.24	XENIX-Compatible Options	109
3.4	Controlling Binary and Text Modes	111

3.1 Introduction

This chapter explains how to compile and link using the **CL** command and discusses commonly used **CL** options. The **CL** command is the only command you need to compile and link your C source files. **CL** executes the three compiler passes, then automatically invokes **LINK**, the Microsoft Overlay Linker, to link your files.

Using the **CL** options described in this chapter, you can control and modify the tasks performed by the command. For example, you can direct **CL** to create an object-listing file or a preprocessed listing. Options also let you give information that applies to the compilation process; you can specify the definitions for manifest (symbolic) constants and macros, and the kinds of warning messages you want to see.

For a quick overview of the more commonly used options, type

```
CL /HELP
```

after the MS-DOS prompt. The **/HELP** option is described in greater detail in Section 3.3.2, "Listing the Compiler Options."

The **CL** command automatically optimizes your program. You never have to give an optimizing instruction unless you either want to change the way **CL** optimizes, request more sophisticated optimizations, or disable optimization altogether. See Section 3.3.13, "Optimizing," for more on these choices.

Section 3.2 explains the basic use of the **CL** command to produce an executable program.

Sections 3.3.1–3.3.24 describe the commonly used **CL** options.

See Chapter 4 for information about linking object files and libraries using the **CL** command. See Chapter 12 of the Microsoft CodeView and Utilities manual for a detailed description of the linker and its options.

See Chapter 6 for a discussion of the **CL** options that control memory models.

See Chapter 7 for a discussion of the **CL** options that control floating-point math operations.

See the *Microsoft C Quick Reference Guide*, provided with this package, for a summary of the **CL** command and its options.

3.2 The Basics: Compiling and Linking C Programs

This section explains how to use **CL** to compile and link C programs and discusses the rules and conventions that apply to file names and options used with **CL**.

3.2.1 The CL Command

The **CL** command has the following form:

```
CL [option]... file... [option... file...] [/link[link-libinfo]]
```

Each *option* is one of the command-line options described in Sections 3.3.1–3.3.24, in Chapter 6, or in Chapter 7.

Each *file* names a source or object file to be processed or a library to be searched at link time. See Section 3.2.1.1 for information about specifying source and object files.

The **CL** command automatically specifies the appropriate library to be used during linking; however, you can use the **/link** option with the optional *link-libinfo* field to specify additional or different libraries, library search paths, and options to be used during linking. You can also specify linker options in the *linkoptions* field. See Section 4.3, “Passing Linker Information: The **/link** Option,” for information about specifying different libraries and linker options.

You can give any number of options, file names, and library names on the command line, provided that the command line does not exceed 128 characters.

3.2.1.1 Specifying Source and Object Files

The **CL** command can process source files, object files, library files, or any combination of these. It uses the file-name extension (the period plus any letters that follow it) to determine what kind of processing the file needs, as shown in the following list:

- If the file has a **.C** extension, **CL** compiles the file.
- If the file has an **.ASM** extension, **CL** displays the following error message to indicate that it cannot invoke the Microsoft Macro Assembler:

```
command-line error D2015: assembly files are not handled
```


- If the file has an **.OBJ** extension, **CL** processes the file by invoking the linker.
- If the file has a **.LIB** extension, **CL** passes the file to the linker to be searched, unless the **/c** option is given to suppress linking. See Section 3.3.4 for a description of the **/c** option.
- If the extension is omitted, **CL** assumes an extension of **.OBJ**. If the extension is anything other than **.C**, **.OBJ**, or **.LIB**, **CL** assumes the file is an object file unless the file name is specified in association with the **/Tc** option. If the file name is specified with the **/Tc** option, **CL** assumes the file is a C source file. See Section 3.3.3 for a description of the **/Tc** option.

■ Examples

```
CL A.C B.C C.OBJ D
```

The command line above compiles the files **A.C** and **B.C**, creating object files named **A.OBJ** and **B.OBJ**. These object files are then linked with **C.OBJ** and **D.OBJ** to form an executable file named **A.EXE** (since the base name of the first file on the command line is **A**). Note that the extension **.OBJ** is assumed for **D** since no extension is given on the command line.

```
CL A.C B.C C.OBJ /TcD.SRC
```

The command line above performs the same operations as the preceding command line, except that the **/Tc** option indicates that **D.SRC** is a source file, not an object file. Thus, the files **A.C**, **B.C**, and **D.SRC** are compiled, creating object files named **A.OBJ**, **B.OBJ**, and **D.OBJ**. These object files are then linked with **C.OBJ** to form an executable file named **A.EXE** (since the base name of the first file on the command line is **A**).

Wild-Card Characters

You can use the MS-DOS wild-card characters (***** and **?**) to process all files whose names match the wild-card specification, if the files have the required extensions. See your DOS user's guide for more information on wild-card characters.

Do not use file names with wild-card characters in **CL** options that take file-name arguments, such as **/Tc**, **/Fo**, and **/I**. Also, do not use wild-card characters in **CL** options given in the **CL** environment variables.

■ Examples

```
CL *.C
```

The command line above compiles all source files with the default extension (**.C**) in the current working directory. The resulting object files are linked to form an executable file whose base name is the same as the base name of the first file compiled.

```
CL *.OBJ
```

The command above links all object files with the default extension (**.OBJ**) in the current working directory.

Path Specifications

Any *filespec* on the **CL** command line can include a full or partial path specification. A full path specification starts with the drive name; a partial path specification gives one or more directory names before the name of the file, but does not give a drive name.

Specifying paths with file names allows you to process files in different directories or on different drives.

Uppercase and Lowercase Letters in File Names

You can use uppercase letters, lowercase letters, or a combination of both for the file names on the **CL** command line. For example, the following three file names are equivalent:

```
abcde.C  
ABCDE.C  
aBcDe.c
```

Note that, unlike file names, **CL** command options *are* case sensitive.

3.2.1.2 Creating Executable Files

When **CL** compiles source files it creates object files. By default, these object files have the same base names as the corresponding source files, but with the extension **.OBJ** instead of **.C**. (The base name of a file extension is the portion of the name preceding the period, but excluding the path specification and drive name, if any.) You can use the **/Fo** option to give a different name to an object file.

Unless the **/c** option is given, **CL** links these object files, along with any **.OBJ** files you give on the command line, to form an executable file. The executable file has the base name of the first file (source or object) given on the command line, plus an **.EXE** extension. If only **.OBJ** files are given on the command line, **CL** skips the compilation stage and simply links the files.

You can tell whether **CL** is compiling or linking by the messages that appear on the screen. When **CL** invokes the compiler, a message similar to the following message appears on your screen:

```
Microsoft (R) C Optimizing Compiler Version 5.00
Copyright (C) Microsoft Corp 1984, 1985, 1986, 1987. All rights reserved.
```

As each source file on the command line is compiled, its name appears on the screen. When all source files have been compiled and the linker is invoked, a message similar to the following message appears:

```
Microsoft (R) Overlay Linker Version 3.60
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.
```

This message is followed by several lines showing Microsoft **LINK** prompts and the responses provided by **CL**.

3.2.1.3 Stopping CL

If you want to stop the compiling and linking session for any reason, press CTRL+C or CTRL+BREAK. You will be returned to the MS-DOS command level. If, after doing this, you discover new files beginning with 00 or 01 in the directory specified by the **TMP** environment variable, you can safely delete them; because the compiling session was interrupted, these temporary compiler files were not deleted.

Certain nonstandard DOS environments (including some commonly used networks) often intercept some or all of the MS-DOS system calls and handle the calls themselves to provide additional or different capabilities. When running the compiler under such environments, the different operation of the system calls may cause **CL** to differ from its documented behavior.

3.2.2 Using the CL Environment Variable

You can also use the **CL** environment variable to specify files and options without giving them on the command line. This variable has the following format:

```
SET CL= [[option]... [filespec]...][ /link[ link-libinfo]]
```

This variable is useful if you usually give a large number of files and options when you compile. Since the files and options that you define with this variable are not counted in the 128-character limit for the command line, you can define the files and options you use most often with the **CL** variable and then give only the files and options you need for specific purposes on the command line.

The information you define in the **CL** variable is treated as though it appeared before the corresponding information you give on the **CL** command line. For example, if you use a command sequence of the form

```
SET CL = opt1 file1 /link link-libinfo1
```

```
CL opt2 file2 /link link-libinfo2
```

the effect would be the same as entering the following **CL** command:

```
CL opt1 file1 opt2 file2 /link link-libinfo1 link-libinfo2
```

Note that if you have given an option in the **CL** environment variable, you generally cannot turn off or change the option from the command line. You must reset the **CL** environment variable and omit the file or option that you do not want to use.

Also note that you cannot use **CL** to set options that use an equal sign (for example, the */D identifier = string* option described in Section 3.3.9.1).

■ Examples

```
SET CL=/Zp /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ
CL INPUT.C
```

In the example above, the **CL** environment variable tells the **CL** command to use the */Zp*, */Ox*, and */I* options during compilation and then link with the object file *\LIB\BINMODE.OBJ*. The **CL** command that follows would then have the same effect as the following command line:

```
CL /Zp /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ INPUT.C
```

That is, it would specify structure packing on two-byte boundaries (Section 3.3.15); perform maximum optimizations (Section 3.3.13.1); search for include files in the *\INCLUDE\MYINCLS* directory (Section 3.3.9.6); and would suppress translation of carriage-return–line-feed character combinations for the source file *INPUT.C* (Section 3.4).

3.3 Using CL Options

The **CL** command offers a large number of command options to control and modify the compiler's operation. Options begin with the forward slash character (/) and contain one or more letters. You can use a dash (-) instead of the forward slash if you prefer. For example, **/Zg** and **-Zg** are both acceptable forms of the **Zg** option. In this manual, forward slashes are used for options, although in error messages dashes are used.

Important

Although file names can be given in either uppercase or lowercase letters, options must be given exactly as shown in this manual. For example, **/W** and **/w** are two different options.

Options can be defined in the **CL** environment variable, or they can appear anywhere on the **CL** command line. In general, an option applies to all files that follow it on the command line, and it does not affect files preceding it on the command line. However, not all options follow this rule; see the discussion of a particular option for information on its behavior. Keep in mind that most **CL** options apply only to the compilation process. Unless specifically noted, options do not affect any object files given on the command line.

Since options defined in the environment are treated as if they appeared before options given on the command line, they affect any files given on the command line. Although, in some cases, conflicting options can be given on the command line to override options defined in the environment, it is usually safer to reset the **CL** variable because conflicts between the environment and command line may cause compilation to fail.

■ Examples

```
SET CL = /FPi87 /AL /Ox
```

```
CL /FPa /AM FILE1.c
```

In the example above, the conflicting floating-point and memory-model options given in the **CL** variable and on the **CL** command line would cause compilation to fail.

The following example illustrates how to turn off the effects of a **CL** option defined in the environment:

```
SET CL = /Za
```

```
CL FILE1.C /Ze FILE2.C
```

In the example above, the **CL** environment variable is set to the **/Za** option, described in Section 3.3.14, which tells the compiler to treat Microsoft extensions to the C language as ordinary identifiers rather than reserved words. The **CL** command specifies the inverse option, **/Ze**, which tells the compiler to treat language extensions as reserved words. Since the effect is the same as compiling with the command line

```
CL /Za FILE1.C /Ze FILE2.C
```

FILE1.C is compiled with language extensions disabled and FILE2.C is compiled with language extensions enabled.

3.3.1 Memory-Model (/A) and Floating-Point (/FP) Options

Two important options that you specify with the **CL** command are the memory model used for your program, and how your program handles floating-point math operations.

You use the **CL** command to specify the memory model your program will use. The memory model defines the rules that the compiler will use to set up the program's code and data segments in memory. **CL** offers the following memory-model options:

Option	Effect
/AS	Chooses the small memory model (default)
/AM	Chooses the medium memory model
/AC	Chooses the compact memory model
/AL	Chooses the large memory model
/AH	Chooses the huge memory model

See Chapter 6 for a description of these options and the memory models they specify.

The **CL** command includes the following options that allow you to choose how the program you are compiling will handle floating-point operations:

Option	Effect
/FPi87	Generates in-line instructions and selects the 8087/80287 math package
/FPi	Generates in-line instructions and selects the emulator math package
/FPc87	Generates floating-point calls and selects the 8087/80287 math package
/FPc	Generates floating-point calls and selects the emulator math package
/FPa	Generates floating-point calls and selects the alternate math package.

See Chapter 7, "Controlling Floating-Point Math Operations," for a description of these options and their effects.

The floating-point and memory-model options you choose determine the name of the standard library that **CL** places in the object file it creates. This library is then considered the default library, since the linker searches for it by default. Table 3.1 shows each combination of memory-model and floating-point options and the corresponding library name that **CL** embeds in the object file.

Table 3.1
CL Options and Default Libraries

Floating-Point Option	Memory-Model Option	Default Library
/FPi87 or /FPc87	/AS	SLIBC7.LIB
	/AM	MLIBC7.LIB
	/AC	CLIBC7.LIB
	/AL or /AH	LLIBC7.LIB
/FPi or /FPc	/AS	SLIBCE.LIB
	/AM	MLIBCE.LIB
	/AC	CLIBCE.LIB
	/AL or /AH	LLIBCE.LIB
/FPa	/AS	SLIBCA.LIB
	/AM	MLIBCA.LIB
	/AC	CLIBCA.LIB
	/AL or /AH	LLIBCA.LIB

Note

If you are linking with any objects compiled with Version 4.0 of Microsoft C, you must explicitly give the **/NOD** (“no default library search”) linker option after the **/link** option on the **CL** command line, then specify the name of the Version 5.0 combined library explicitly.

3.3.2 Listing the Compiler Options (/HELP)

■ Option

/HELP
/help

This option displays a list of the most commonly used compiler options. **CL** processes all information on the line containing the **/help** option, and displays the command list.

This option is not case sensitive: any combination of uppercase and lowercase letters is acceptable. For example, **/hELp** is a valid form of this option.

If you specify the **/qc** option (described in Section 3.3.5) before the **/HELP** option on the command line, only the options that work under **/qc** are displayed.

3.3.3 Specifying Source Files (/Tc)

■ Option

/Tc sourcefile

The **/Tc** option tells the **CL** command that the given file is a C source file. One or more spaces can appear between **/Tc** and the source-file name.

If this option does not appear, **CL** assumes that files with the extension **.C** are C source files, files with the extension **.LIB** are libraries, and files with any other extension or with no extension are object files. If you use the **/Tc** option, **CL** treats the given file as a C source file, regardless of its extension, if any. A separate **/Tc** option must appear for each source file that has an extension other than **.C**.

If you have to specify more than one source file with an extension other than **.C**, you must specify each source file in a separate **/Tc** option.

■ Example

```
CL MAIN.C /Tc TEST.PRG /Tc COLLATE.PRG PRINT.PRG
```

In the example above, the **CL** command compiles the three source files MAIN.C, TEST.PRG, and COLLATE.PRG. Since the file PRINT.PRG is given without a **/Tc** option, **CL** treats it as an object file. Thus, after compiling the three source files, **CL** links the object files MAIN.OBJ, TEST.OBJ, COLLATE.OBJ, and PRINT.PRG.

3.3.4 Compiling without Linking (/c)

■ Option

/c

The **/c** (for “compile-only”) option suppresses linking. Source files given on the command line are compiled, but the resulting object files are not linked, no executable file is created, and any object files specified on the command line are ignored. This option is useful when you are compiling individual source files that do not make up a complete program.

The **/c** option applies to the entire **CL** command line, regardless of the option’s position in the command line.

■ Example

```
CL /c *.C
```

This command line compiles, but does not link, all files with the extension **.C** in the current working directory.

3.3.5 Compiling with QuickC (/qc, /Zr, /Zq)

■ Option

/qc

The **/qc** option tells the compiler to compile any source files specified on the remainder of the command line using the Microsoft QuickC Compiler. Because only limited optimizations are performed, programs produced using this option are generally slower and larger than programs produced without it. However, they can be compiled much faster using this option.

If you give the **/qc** option, only the following **CL** options have any effect on the program:

/A	/Fm	/help	/Tc	/X
/c	/Fo	/I	/U	/Zs
/D	/FPi	/J	/u	/Ze
/F	/Gs	/link	/W	/Zp
/Fc	/Gt	/P	/w	/Zs

The following options affect compilation only if you also specify the **/qc** option; otherwise, the **CL** command ignores them:

Option	Effect
/Zr	Checks for null pointers and out-of-range far pointers at run time
/Zq	Generates debugging interrupts for programs that will be debugged within the QuickC environment

Also, the following features are illegal or have no effect in source programs compiled with the **/qc** option:

- Use of the **huge** keyword, which is ignored if language extensions are enabled (that is, if the program is compiled with the default language-extensions option, **/Ze**)
- Redclaration of **extern** items as **static** items, which causes a redefinition error
- Use of the **loop_opt**, **intrinsic**, **function**, **alloc_text**, and **same_seg** pragmas

Refer to the *Microsoft QuickC Programmer's Guide* provided in this package for complete documentation of the Microsoft QuickC Compiler.

3.3.6 Naming the Object File (/Fo)

■ Option

/Foobjfile

By default, **CL** gives each object file it creates the base name of the corresponding source file plus the extension **.OBJ**. The **/Fo** option lets you give different names to object files or create them in a different directory. If you are compiling more than one source file, you can give an **/Fo** option for each source file to rename the corresponding object file.

Keep the following rules in mind when using this option:

- The *objfile* argument must appear immediately after the option, with no intervening spaces.
- Each **/Fo** option applies to the next source file that appears on the command line after the option.

You are free to supply any name and any extension you like for the *objfile*. However, it is recommended that you use the conventional **.OBJ** extension because the linker and the **LIB** library manager use **.OBJ** as the default extension when processing object files.

If you do not give a complete object file name with the **/Fo** option (that is, if you do not give an object file name with a base and an extension), **CL** names the object files according to the following rules:

- If you give an object-file name without an extension (such as **TEST**), **CL** automatically appends the **.OBJ** extension.
- If you give an object-file name with a blank extension (such as **TEST.**), **CL** leaves the extension blank.
- If you give only a drive or directory specification following the **/Fo** option, **CL** creates the object file on the given drive or directory and uses the default file name (the base name of the source file plus **.OBJ**).

You can use this option to create the object file in another directory or on another disk. When you give only a directory specification, the directory specification must end with a backslash (\) so that **CL** can distinguish between a directory specification and a file name.

■ Examples

```
CL /FoB:\OBJECT\ THIS.C
```

In the example above, the source file **THIS.C** is compiled; the resulting object file is named **THIS.OBJ** (by default). The directory specification **B:\OBJECT** tells **CL** to create **THIS.OBJ** in the directory named **\OBJECT** on Drive B.

```
CL /Fo\OBJECT\ THIS.C THAT.C /Fo\src\NEWTROSE.OBJ THOSE.C
```

In the example above, the first **/Fo** option tells the compiler to create, in the **\OBJECT** directory, the object files **THIS.OBJ** (created as a result of compiling **THIS.C**) and **THAT.OBJ** (created as a result of compiling

THAT.C). The second **/Fo** option tells the compiler to create the object file named NEWTHOSE.OBJ, (created as a result of compiling THOSE.C) in the \SRC directory.

3.3.7 Naming the Executable File (/Fe)

■ Option

/Feexefile

By default, **CL** gives the base name of the first file (source or object) on the command line, plus the extension **.EXE**, to the executable file it creates. The **/Fe** option lets you give the executable file a different name or create it in a different directory.

Since **CL** creates only one executable file, you can give the **/Fe** option anywhere on the command line. If more than one **/Fe** option appears, **CL** gives the executable file the name specified in the last **/Fe** option on the command line.

The **/Fe** option applies only in the linking stage. If you specify the **/c** option to suppress linking, **/Fe** has no effect.

The *exefile* argument must appear immediately after the option, with no intervening spaces. The *exefile* argument can be a file specification, a drive name, or a path specification. If *exefile* is a drive name or path specification, the **CL** command creates the executable file in the given location, using the default name (base name of the first file plus **.EXE**). When you give a path specification as the *exefile* argument, the path specification must end with a backslash (\) so that **CL** can distinguish it from an ordinary file name.

You are free to supply any name and any extension you like for the *exefile*. If you give a file name without an extension, **CL** automatically appends the **.EXE** extension.

■ Examples

```
CL /FeC:\BIN\PROCESS *.C
```

The example above compiles and links all source files with the extension **.C** in the current working directory. The resulting executable file is named **PROCESS.EXE** and is created in the directory **C:\BIN**.


```
CL /FeC:\BIN\ *.C
```

The example above is similar to the first example except that the executable file, instead of being named `PROCESS.EXE`, is given the same base name as the first file compiled. The executable file is created in the directory `C:\BIN`.

3.3.8 Creating Listing Files

A number of listing options are available with the **CL** command. You can create a source listing, a map listing, or one of several kinds of object listings. You can also set title and subtitle of the source listing from the command line and control the length of source-listing lines and pages.

The options available for producing listings and controlling their appearance are described in the following sections.

Note

Listings produced by the **CL** command may contain names that begin with more than one underscore (for example, `__chkstk`) or that end with the suffix `QQ`. Names that use these conventions are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *Microsoft C Run-Time Library Reference* such as `_psp`, `_amblksiz`, and `_fpreset()`. Moreover, you should avoid creating global names that begin with an underscore. Since the compiler automatically adds another leading underscore, these names will have two leading underscores and might conflict with the names reserved by the compiler.

3.3.8.1 Types of Listings (/Fs, /Fl, /Fa, /Fc, /Fm)

■ Options

<code>/Fs</code> <i>[[listfiles]]</i>	Source listing
<code>/Fl</code> <i>[[listfile]]</i>	Object listing
<code>/Fa</code> <i>[[listfile]]</i>	Assembly listing
<code>/Fc</code> <i>[[listfile]]</i>	Combined source and object listing
<code>/Fm</code> <i>[[mapfile]]</i>	Map file that lists segments, in order

This section describes how to use command-line options to create listings. For an example of each type of listing and a description of the information it contains, see Section 3.3.8.5, “Formats for Listings.”

When using the options described in this section, the *listfile* argument, if given, must follow the option immediately, with no intervening spaces. The *listfile* can be a file specification, a drive name, or a path specification. It can also be omitted.

Important

When you give just a path specification as the *listfile* argument, the path specification must end with a backslash (\) so that **CL** can distinguish it from an ordinary file name.

When you give a drive name or path specification as the argument to a listing option, or if you omit the argument altogether, **CL** uses the default file name for the listing type. Table 3.2 gives the default names used for each type of listing. The table also shows the default extensions, which are used when you give a file-name argument that lacks an extension.

Table 3.2
Default File Names and Extensions

Option	Listing Type	Default File Name ¹	Default Extension ²
/Fs	Source	Base name of source file plus .LST	.LST
/Fl	Object	Base name of source file plus .COD	.COD
/Fa	Assembly	Base name of source file plus .ASM	.ASM
/Fc	Combined source-object	Base name of source file plus .COD	.COD
/Fm	Map	Base name of first source or object file on the command line plus .MAP	.MAP

¹ The default file name is used when the option is given with no argument or with a drive name or path specification as the argument.

² The default extension is used when a file name lacking an extension is given.

Since you can process more than one file at a time with the **CL** command, the order in which you give listing options and the kind of argument you

give for each option (file specification, path specification, or drive name) affect the result. Table 3.3 summarizes the effects of each option with each type of argument.

Table 3.3
Arguments to Listing Options

Option	File-Name Argument	Drive-Name or Path Argument ¹	No Argument
/Fa , /Fc , /Fl , /Fs	Creates a listing for next source file on command line; uses default extension if no extension is supplied	Creates listings in the given location for every source file listed after the option on the command line; uses default names	Creates listings in the current directory for every source file listed after the option on the command line; uses default names
/Fm	Uses given file name for the map file; uses default extension if no extension is supplied	Creates map file in the given directory; uses default name	Uses default name

¹ When you give just a path specification as the argument, the path specification must end with a backslash (\) so that **CL** can distinguish it from an ordinary file name.

Only one type of object or assembly listing can be produced for each source file. The **/Fc** option overrides the **/Fa** and **/Fl** options; whenever you use **/Fc**, a combined listing is produced. If you apply both the **/Fa** and the **/Fl** options to one source file, only the last listing specified on the command line is produced. If you specify both the **/Fa** and the **/Fs** options to one source file, a combined listing is produced.

Note

The **CL** command optimizes by default, so listing files reflect the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear, especially when you use the **/Fc** option to mingle the source and assembly codes. To produce a listing without optimizing, use the **/Od** option (discussed in Section 3.3.12, "Preparing for Debugging") with the listing option.

The map file is produced during the linking stage. If linking is suppressed with the `/c` option, the `/Fm` option has no effect.

■ Examples

```
CL /FsHELLO.SRC /FcHELLO.CMB HELLO.C
```

In the first example, **CL** creates a source listing called `HELLO.SRC` and a combined source and assembly listing called `HELLO.CMB`. The object file has the default name `HELLO.OBJ`.

```
CL /FsHELLO.SRC /FsHELLO.LST /FcHELLO.COD HELLO.C
```

The example above produces a source listing called `HELLO.LST` rather than `HELLO.SRC`, since the last name provided has precedence. This example also produces an object-listing file named `HELLO.COD`. The object file in this example has the default name `HELLO.OBJ`.

3.3.8.2 Special File Names

You can use the MS-DOS device names listed below as file-name arguments to the listing options. These special names allow you to direct listing files to your terminal or printer:

Name	Device
AUX	Refers to an auxiliary device.
CON	Refers to the console (terminal).
PRN	Refers to the printer device.
NUL	Specifies a "null" (nonexistent) file. Giving NUL as a file name means that no file is created.

Even if you add device designations or file-name extensions to these special file names, they remain associated with the devices listed above. For example, `A:CON.XXX` still refers to the console and is not a disk-file name.

Note

When using these device names, do not append a colon. The Microsoft C Optimizing Compiler does not recognize the colon. For example, use CON or PRN, not CON: or PRN:.

3.3.8.3 Setting Line Width (/Sl) and Page Length (/Sp)

■ Options

/Sl linewidth
/Sp pagelength

The */Sl* and */Sp* options let you change the line width and page length, respectively, for source listings. These options are useful in preparing source listings for a printer that uses nonstandard page lengths. The space is optional between */Sl* and *linewidth* or */Sp* and *pagelength*.

The *linewidth* argument gives the width of the listing line in columns (on line printers, columns usually correspond to characters). The number given must be a positive integer between 79 and 132, inclusive. If you specify any number outside this range, the compiler generates a diagnostic message and uses the default line width (79 columns). Any line that exceeds the listing width is truncated.

The *pagelength* argument gives the number of lines to appear on each page of the listing. The number given must be a positive integer between 15 and 255, inclusive. If you specify any number outside this range, the compiler generates a diagnostic message and uses the default page length (63 lines).

The */Sl* or */Sp* option applies to the remainder of the command line or until the next occurrence of */Sl* or */Sp* on the command line. These options do not create source listings; they take effect only if you also specify the */Fs* option to create a source listing.

■ Example

```
CL /c /Fs /Sl 90 /Sp 70 *.C
```

The example above compiles all C source files with the default extension (.C) in the current working directory, creating a source-listing file for each source file. Each page of the source-listing file is 90 columns wide and 70 lines long.

3.3.8.4 Setting Titles (/St) and Subtitles (/Ss)

■ Options

```
/St "title"  
/Ss "subtitle"
```

The /St and /Ss options set the title and subtitle, respectively, for source listings. The quotation marks (" ") around the *title* or *subtitle* argument can be omitted if the title or subtitle does not contain space or tab characters. The space between /St or /Ss and their arguments is optional.

The title appears in the upper left corner of each page of the source listing. The subtitle appears below the title.

The /St or /Ss option applies to the remainder of the command line or until the next occurrence of /St or /Ss on the command line. These options do not cause source listings to be created. They take effect only when the /Fs option is also used to create a source listing.

■ Examples

```
CL /St "INCOME TAX" /Ss 4-14 /Fs TAX*.C
```

The example above compiles and links all source files beginning with TAX and ending with the default extension (.C) in the current working directory. Each page of the source listing contains the title INCOME TAX in the upper left corner. The subtitle 4-14 appears below the title on each page.

```
CL /c /Fs /St"CALC PROG" /Ss"COUNT" CT.C /Ss"SORT" SRT.C
```

The example above compiles two source files and creates two source listings. Each source listing has a unique subtitle, but both listings have the title CALC PROG.

3.3.8.5 Formats for Listings

This section describes and shows examples of the five types of listings available with the **CL** command. See Section 3.3.8.1, "Types of Listings," for information on how to create these listings.

Source Listing

Source listings are helpful in debugging programs as they are being developed. These listings are also useful for documenting the structure of a finished program.

The source listing contains the numbered source-code lines of each procedure in the source file, along with any diagnostic messages that were generated. If the source file compiles with no errors more serious than warning errors, the source listing also includes tables of local symbols, global symbols, and parameter symbols for each function. If the compiler is unable to finish compilation, it does not generate symbol tables.

At the end of the source listing is a summary of the segment sizes in your program. This summary is useful for analyzing the memory requirements of your program.

Any error messages that occurred during compilation appear in the listing after the line that caused the error, as shown in the following example:

```

1 char hexvalue[10];
2
3 main()
4 {
5     long htoi();
6     printf("Please enter the hex value you want to convert:\n");
7     scanf("%s", hexvalue);
8     printf("The integer value of the hex value is %ld\n", hexvalue));
9 }
10
11 long htoi(hexvalue)
12 char *hexvalue;
13 {
14     register char *ptr=hexvalue;
15     int i=0;
16     long n=0;
17     long exp16();
18     while (*ptr != '\0') {
19         if (*ptr >= 'a' && *ptr <= 'f')
20             *ptr -= 87;
21         else if (*ptr >= 'A' && *ptr <= 'F')
22             *ptr -= 55;
23         else
24             *ptr -= 48;
25         ptr++;
26     }
bomb.c(25) : error C2059: syntax error : ';'

```

The line number given in the error message corresponds to the number of the source line immediately above the message in the source listing.

The following example shows the source listing for a simple C program:

Hex to ASCII
2/25/87

PAGE 1
02-25-87
10:44:23

Line# Source Line

Microsoft C Compiler Version 5.00

```
1 char hexvalue[10];
2
3 main()
4 {
5     long htoi();
6     printf("Please enter the hex value you want to convert:");
7     scanf("%s", hexvalue);
8     printf("The integer value of the hex value is %ld\n", htoi(hexvalue));
9 }
10
11 long htoi(hexvalue)
12 char *hexvalue;
13 {
14     register char *ptr=hexvalue;
15     int i=0;
16     long n=0;
17     long exp16();
18     while (*ptr != '\0') {
19         if (*ptr >= 'a' && *ptr <= 'f')
20             *ptr -= 87;
21         else if (*ptr >= 'A' && *ptr <= 'F')
22             *ptr -= 55;
23         else
24             *ptr -= 48;
25         ptr++;
26     }
27     ptr -= 1;
28     while (ptr < hexvalue)
29     {
30         n += (*ptr * exp16(i));
31         i++;
32         ptr++;
33     }
34     return(n);
35 }
```

htoi Local Symbols

Name	Class	Type	Size	Offset	Register
i	auto			-0008	
ptr	auto			***	si
n	auto			-0004	
hexvalue.	param			0004	

```
35 }
36
37 long exp16(exp)
38 int exp;
39 {
40     long result=1;
41     int j;
42     for (j=1; j<=exp; j++)
43         result *= 16;
44     return(result);
45 }
```


Hex to A
2/25/87

02-25-87
10:44:23

Microsoft C Compiler Version 5.00

exp16 Local Symbols

Name	Class	Type	Size	Offset	Register
j	auto			-0006	
result.	auto			-0004	
exp	param			0004	

Global Symbols

Name	Class	Type	Size	Offset
exp16	global	near function	***	00ae
hexvalue.	common	struct/array	10	***
htoi.	global	near function	***	0038
main.	global	near function	***	0000
printf.	extern	near function	***	***
scanf	extern	near function	***	***

Code size = 00e8 (232)
Data size = 005f (95)
Bss size = 0000 (0)

No errors detected

At the end of each function, a table of local symbols is given, as shown below for the function htoi:

htoi Local Symbols

Name	Class	Type	Size	Offset	Register
i	auto			-0008	
ptr	auto			***	si
n	auto			-0004	
hexvalue.	param			0004	

The following list shows the contents of each column:

Column	Contents
Name	The name of each local symbol in the function.
Class	Either auto if the symbol is a nonstatic local variable, or param if the symbol is a formal parameter.
Offset	The symbol's offset address relative to the frame pointer (that is, the BP register). The Offset number is positive for param symbols and negative for auto symbols with auto storage class.
Register	Blank unless the variable is stored in a register; if the variable is stored in a register, this column indicates the register (SI or DI).

At the end of the source code, a table of global symbols is given, as shown below:

Name	Class	Type	Size	Offset
exp16	global	near function	***	00ae
hexvalue.	common	struct/array	10	***
htoi.	global	near function	***	0038
main.	global	near function	***	0000
printf.	extern	near function	***	***
scanf	extern	near function	***	***

The following list shows the contents of each column:

Column	Contents
Name	Each global symbol, external symbol, and statically allocated variable declared in the source file.
Class	Either <code>global</code> , <code>common</code> , <code>extern</code> , or <code>static</code> , depending on how the symbol was defined in the source file.
Type	<p>A simplified version of the symbol's type as declared in the source file.</p> <p>For functions, this entry is either <code>near function</code> or <code>far function</code>, depending on which memory model was used and how the function was declared. For a pointer, this entry is <code>near pointer</code>, <code>far pointer</code>, or <code>huge pointer</code>. For enumeration variables, this entry is <code>int</code>. For structures, unions, and arrays, this entry is <code>struct/array</code>.</p>
Size	Used only for variables. Specifies the number of bytes of storage allocated for the variable. Since the amount of storage allocated for an external array may not be known, its <code>Size</code> entry may be undefined.
Offset	<p>Used only for symbols with an entry of <code>global</code> or <code>static</code> in the <code>Class</code> column.</p> <p>For variables, this entry gives the relative offset of the variable's storage in the logical data segment for the program file being compiled. Since the linker usually combines several logical data segments into a physical segment, this number is useful only for determining the relative position of storage of variables. For functions, this entry gives the relative offset of the start of the function in the logical code segment. For small-model programs, the linker combines logical code into a single</p>

physical segment, so this entry is useful for determining the relative positions of different functions defined in the same source file. However, for medium-, large-, and huge-model programs, each logical code segment becomes a unique physical segment. In these cases, this entry gives the actual offset of the function in its run-time code segment.

The last table in the source listing shows the segments used and their size, as shown below:

```
Code size = 0103 (259)
Data size = 005f (95)
Bss size  = 0000 (0)
```

The number of bytes in each segment is given first in hexadecimal, and then in decimal (in parentheses).

Object Listing

The **/Fl** option produces an object listing. The object listing contains the instruction encoding and assembly code for your program. The line numbers are shown in the listing as comments. The instruction encoding is on the left and assembly code on the right, as shown in the sample below:

```
; Line 4
_main      PUBLIC  _main
           PROC NEAR
           *** 000000      55                      push bp
           *** 000001      8b ec                    mov  bp,sp
           *** 000003      33 c0                    xor  ax,ax
           *** 000005      e8 00 00                  call __chkstk
; Line 6
           *** 000008      b8 00 00                  mov  ax,OFFSET DGROUP:$S G12
           *** 00000b      50                      push ax
           *** 00000c      e8 00 00                  call _printf
           *** 00000f      83 c4 02                  add  sp,2
```

Assembly Listing

The **/Fa** option produces an assembly listing. The assembly listing contains the assembly code corresponding to your C source file, as shown below:

```

; Line 4
_main      PUBLIC  _main
           PROC NEAR
           push    bp
           mov     bp,sp
           xor     ax,ax
           call    __chkstk
; Line 6
           mov     ax,OFFSET DGROUP:$SG12
           push    ax
           call    _printf
           add     sp,2

```

Note that the sample shows the same code as in the object listing sample, except that the instruction encoding is omitted.

The listing generated by the **/Fa** option in Versions 5.0 and later of the Microsoft C Optimizing Compiler can be used as input to the Microsoft Macro Assembler (**MASM**).

Combined Source and Object Listing

The **/Fc** option produces a combined source and object listing. The combined source and object listing shows each line of your source program followed by the corresponding line (or lines) of machine instructions, as in the following sample:

```

_TEXT      SEGMENT
;|*** char hexvalue[10];
;|***
;|*** main()
;|*** {
; Line 4
_main      PUBLIC  _main
           PROC NEAR
           *** 000000      55                      push bp
           *** 000001      8b ec                    mov  bp,sp
           *** 000003      33 c0                    xor  ax,ax
           *** 000005      e8 00 00                  call __chkstk
;|*** long htoi();
;|*** printf("Please enter the hex value you want to convert:0);
; Line 6
           *** 000008      b8 00 00                  mov  ax,OFFSET DGROUP:$SG12
           *** 00000b      50                      push ax
           *** 00000c      e8 00 00                  call _printf
           *** 00000f      83 c4 02                  add  sp,2
;|*** scanf("%s", hexvalue);

```

Note that this sample is like the object-listing sample, except that the source-program line is provided in addition to the line number.

When you examine a listing file, you will notice that the names of globally visible functions and variables begin with an underscore, as shown below (this part of the listing is the same for all three kinds of listings):


```

EXTRN  _printf:NEAR
EXTRN  _scanf:NEAR
EXTRN  __chkstk:NEAR
EXTRN  __aInmul:NEAR
EXTRN  __aNNalshl:NEAR
EXTRN  _hexvalue:TBYTE

```

The Microsoft C Optimizing Compiler automatically prefixes an underscore to all global names to preserve compatibility with XENIX C compilers. If you write assembly-language routines to interface with your C program, this naming convention is important; see Section 3.3.8 for more information.

The listing may also contain names that begin with more than one underscore (for example, `__chkstk` in the sample). Identifiers with more than one leading underscore are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *Microsoft C Run-Time Library Reference* such as `_psp`, `_amblksiz`, and `_fpreset()`. Moreover, you should avoid creating global names that begin with an underscore. Since the compiler automatically adds another leading underscore, these names will have two leading underscores and might conflict with the names reserved by the compiler.

Map File

The `/Fm` option produces a map file. The map file contains a list of segments in order of their appearance within the load module. An example is shown below:

Start	Stop	Length	Name	Class
00000H	01E9FH	01EAOH	_TEXT	CODE
01EAOH	01EAOH	00000H	C_ETEXT	ENDCODE
.

The information in the `Start` and `Stop` columns shows the 20-bit address (in hexadecimal) of each segment, relative to the beginning of the load module. The load module begins at location zero. The `Length` column gives the length of the segment in bytes. The `Name` column gives the name of the segment, and the `Class` column gives information about the segment type. See Chapter 12 of the *Microsoft CodeView and Utilities* manual for information about groups, segments, and classes.

The starting address and name of each group appear after the list of segments. A sample group listing is shown below:

Origin	Group
01EA:0	DGROUP

In the example above, **DGROUP** is the name of the data group. **DGROUP** is the only group used for data segments by programs compiled with the Microsoft C Optimizing Compiler, Versions 4.0 and 5.0.

The map file shown below contains two lists of global symbols: the first list is sorted in ASCII-character order by symbol name and the second is by symbol address. A maximum of 2048 symbols are sorted in each list. (To increase the number of sorted symbols, you must specify the **/MAP** linker option with the *number* argument to create the map file; see Section 4.4 for details.) The notation **Abs** appears next to the names of absolute symbols (symbols containing 16-bit constant values that are not associated with program addresses).

Many of the global symbols that appear in the map file are symbols used internally by the Microsoft C Optimizing Compiler. These symbols usually begin with one or more leading underscores or end with QQ.

Address		Publics by Name
01EA:0096		STKHQQ
0000:1D86		_brkctl
01EA:04B0		_edata
01EA:0910		_end
.		
.		
.		
01EA:00EC		__abrkp
01EA:009C		__abrktb
01EA:00EC		__abrktbe
0000:9876	Abs	__acrtmsg
0000:9876	Abs	__acrtused
.		
.		
.		
01EA:0240		___argc
01EA:0242		___argv
Address		Publics by Value
0000:0010		_main
0000:0047		_atoi
0000:00DA		_exp16
0000:0113		__chkstk
0000:0129		__astart
0000:01C5		__cintDIV
.		
.		
.		

The addresses of the external symbols are in the "*frame:offset*" format, showing the location of the symbol relative to zero (the beginning of the load module).

Following the lists of symbols, the map file gives the program entry point, as shown in the following example:

Program entry point at 0000:0129

3.3.9 Controlling the Preprocessor

The **CL** command provides several options that control the operation of the C preprocessor. You can define macros and manifest (symbolic) constants from the command line, change the search path for include files, and stop compilation of a source file after the preprocessing stage to produce a preprocessed source-file listing. The options that perform these tasks are described in Sections 3.3.9.1–3.3.9.4.

The C preprocessor recognizes only preprocessor directives. It treats the source file as a text file, processing substitutions and definitions as directed. The preprocessor can be run on a file at any stage of development, whether or not the file is a complete C source file. In fact, the preprocessor is not restricted to processing C files; it can be run on any kind of file. However, input files to the preprocessor must follow the preprocessor rules; therefore, not all arbitrary text files may be suitable for use with the preprocessor. See Chapter 8 of the *Microsoft C Quick Reference Guide* for a complete discussion of C preprocessor directives and the format expected for preprocessor input.

3.3.9.1 Defining Constants and Macros (/D)

■ Option

/D *identifier*[= [*string*]]

The **/D** option lets you define a constant or macro used in your source file. The *identifier* is the name of the constant or macro and *string* is its value or meaning. Note that spaces are permitted (but not required) between **/D** and the identifier.

If you leave out both the equal sign and *string*, the given constant or macro is assumed to be defined, and its value is set to 1. For example, **/DSET** is sufficient to define SET.

If you give the equal sign with an empty string, the given constant or macro is considered defined; its definition is the empty string. This definition effectively removes all occurrences of the identifier from the source file. For example, to remove all occurrences of `register`, use the following option:

/Dregister=

Note that the identifier `register` is still considered to be defined.

Note

The `/D identifier` form of this option can be defined using the `CL` environment variable; however, the `/D identifier =` and `/D identifier = string` forms cannot.

The effect of using the `/D` option is the same as using a preprocessor `#define` directive at the beginning of your source file: the identifier is defined in the source file being compiled either until an `#undef` directive removes the definition or until the end of the file is reached.

You can supply a command-line definition for an identifier that is also defined within the source file. However, you must use `#undef` to remove the source-file definition, unless the source-file definition is identical to the command-line definition. The command-line definition remains in effect until the identifier is removed with an `#undef` directive.

Normally, up to 17 definitions are allowed on the command line. Using either the `/Za` option or the `/J` option on the command line reduces to 16 the number of definitions allowed; using both of these options reduces the number to 15. If you need to define more than the maximum number of identifiers, you can remove certain predefined definitions from the command line; see the discussion of the `/U` and `/u` options in Section 3.3.9.3, "Removing Definitions of Predefined Identifiers," for more information.

The `/D` option is especially useful with the `#if` and `#ifdef` directives because you can control conditional-compilation directives in the source file from the command line.

■ Examples

```
CL /D NEED=2 MAIN.C
```

The example above defines the manifest constant `NEED` in the source file `MAIN.C`. This definition is equivalent to placing the directive

```
#define NEED 2
```

at the top of the source file.

For the next example, suppose a source file named `OTHER.C` contains the following fragment:

```
#if defined(NEED)
.
.
.
#endif
```

Suppose further that `OTHER.C` does not explicitly define `NEED` (that is, no `#define` directive for `NEED` is present). Then all statements between the `#if` and the `#endif` directives are compiled only if you supply a definition of `NEED` by using `/D`. For instance, the command

```
CL /DNEED MAIN.C
```

is sufficient to compile all statements following the `#if` directive. Note that `NEED` does not have to be set to a specific value to be considered defined. The following command, in contrast, causes the statements in the `#if` block to be ignored (not compiled):

```
CL MAIN.C
```

3.3.9.2 Predefined Identifiers

The compiler defines four identifiers that are useful in writing portable programs. You can use these identifiers to conditionally compile code sections, depending on the processor and operating system being used. The predefined identifiers and their functions are listed below:

Identifier	Function
MSDOS	Always defined. Identifies target operating system as MS-DOS.
M_I86	Always defined. Identifies target machine as a member of the I86 family.
M_I86mM	Always defined. Identifies memory model, where <i>m</i> is either S (small model), C (compact model), M (medium model), L (large model), or H (huge model). If huge model is used, both M_I86LM and M_I86HM are defined. Small model is the default. Memory models are discussed in Chapter 6, "Working with Memory Models."

NO_EXT_KEYS

Defined only when the **/Za** option is given, thus disabling Microsoft-specific language extensions and extended keywords. See Section 3.3.14, "Enabling and Disabling Language Extensions," for more information.

_CHAR_UNSIGNED

Defined only when the **/J** option is given to make the **char** type unsigned by default. See Section 3.3.20, "Changing the Default char Type," for more information.

3.3.9.3 Removing Definitions of Predefined Identifiers (/U, /u)

■ Options

/U identifier
/u

The **/U** (for "undefine") option turns off the definition of one of the predefined identifiers discussed in the previous section; one or more spaces may separate the **/U** and *identifier*. You can specify more than one **/U** option on the same command line. The **/u** option turns off all four definitions.

These options are useful if you want to give more than the maximum number of definitions (16 if the **/Za** or **/J** option is used, 15 if both options are given, or 17 otherwise) on the command line, or if you have other uses for the predefined identifiers. For each definition of a predefined identifier you remove, you can substitute a definition of your own on the command line. When the definitions of all four predefined identifiers are removed, you can specify up to 20 command-line definitions. However, note that MS-DOS limits the number of characters you can type on a command line, so the number of definitions you can specify in practice is probably fewer than 20.

■ Example

```
CL /UMSDOS /UM_I86 WORK.C;
```

This example removes the definitions of two predefined identifiers. Note that the **/U** option must be given twice to do this.

3.3.9.4 Producing a Preprocessed Listing (/P, /E, /EP)

■ Options

/P Writes preprocessed output to a file
/E Writes preprocessed output to standard output; includes **# line** directives
/EP Writes preprocessed output to a file and standard output

The **/P**, **/E**, and **/EP** options produce listings of preprocessed files. These options allow you to examine the output of the C preprocessor.

The preprocessed listing file is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. All three options suppress compilation; no object file or listing is produced, even if you specify an **/Fo** option or a listing-file option on the **CL** command line.

The **/P** option writes the preprocessed listing to a file with the same base name as the source file, but with an **.I** extension.

The **/E** option copies the preprocessed listing to the standard output (usually your terminal). It places a **# line** directive in the output at the beginning and end of each included file and around lines removed by preprocessor commands that specify conditional compilation. You can use MS-DOS redirection to save this output in a disk file.

The **/E** option is useful when you want to resubmit the preprocessed listing for compilation. The **# line** directives renumber the lines of the preprocessed file so that errors generated in later stages of processing refer to the original source file rather than to the preprocessed file.

Using the **/EP** option combines features of the **/E** and **/P** options; the file is preprocessed and copied to the standard output, but no **# line** directives are added.

■ Examples

```
CL /P MAIN.C
```

The example above creates the preprocessed file **MAIN.I** from the source file **MAIN.C**.

```
CL /E ADD.C > PREADD.C
```

The command above creates a preprocessed file with inserted **#line** directives from the source file `ADD.C`. The output is redirected to the file `PREADD.C`.

```
CL /EP ADD.C
```

The command above produces the same preprocessed output as the second example, but without the **#line** directives. The output appears on the screen.

3.3.9.5 Preserving Comments (/C)

■ Option

`/C`

Normally, comments are stripped from a source file in the preprocessing stage, since they do not serve any purpose in later stages of compiling. The `/C` (for “comment”) option preserves comments during preprocessing. The `/C` option is valid only when the `/E`, `/P`, or `/EP` option is also used.

■ Example

```
CL /P /C SAMPLE.C
```

The example produces a listing named `SAMPLE.I`. The listing file contains the original source file, including comments, with all preprocessor directives expanded or replaced.

3.3.9.6 Searching for Include Files (/I, /X)

■ Options

`/I` *directory*

`/X`

The `/I` and `/X` options temporarily override or change the effects of the environment variable **INCLUDE**. These options let you give a particular file special handling without changing the compiler environment you normally use. (See Section 2.4.5, “Setting Up the Environment,” for a discussion of environment variables.)

You can add to the list of directories searched for include files by using the **/I** (for “include”) option. This option causes the compiler to search the directory or directories you specify before searching the standard places given by the **INCLUDE** environment variable. The space between **/I** and *directory* is optional. You can add more than one include directory by giving the **/I** option more than once in the **CL** command. The directories are searched in order of their appearance in the command line.

The directories are searched only until the specified include file is found. If the file is not found in the given directories or the standard places, the compiler prints an error message and stops processing. When this occurs, you must restart compilation with a corrected directory specification.

You can prevent the C compiler from searching the standard places for include files by using the **/X** (for “exclude”) option. When **CL** sees the **/X** option, it considers the list of standard places to be empty. This option is often used with the **/I** option to define the location of include files that have the same names as include files found in other directories, but that contain different definitions.

■ Examples

```
CL /I \INCLUDE /I\MY\INCLUDE MAIN.C
```

In the example above, **CL** looks for the include files requested by **MAIN.C** in the following order: first in the directory **\INCLUDE**, then in the directory **\MY\INCLUDE**, and finally in the directory or directories assigned to the **INCLUDE** environment variable.

```
CL /X /I \ALT\INCLUDE MAIN.C
```

In the example above, the compiler looks for include files only in the directory **\ALT\INCLUDE**. First the **/X** option tells **CL** to consider the list of standard places empty; then the **/I** option specifies one directory to be searched.

3.3.10 Using the 80186, 80188, or 80286 Processor (**/G0**, **/G1**, **/G2**)

■ Options

- /G0** Enables instruction set for 8086/8088 processor (default)
- /G1** Enables instruction set for 80186/80188 processor
- /G2** Enables instruction set for 80286 processor

If you have an 80186, 80188, or 80286 processor, you can use the **/G1** or **/G2** option to enable the instruction set for your processor. Use **/G1** for the 80186 and 80188 processors; use **/G2** for the 80286. Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but you want your code to be able to run on an 8086, you should not use the 80186/80188 or 80286 instruction set.

The **/G0** option enables the instruction set for the 8086/8088 processor. You do not have to specify this option explicitly, since the 8086/8088 instruction set is used by default. Programs compiled this way will also run on the machines with the 80186, 80188, or 80286 processor.

3.3.11 Checking for Program Errors

You may encounter several different kinds of error messages when you compile, link, and run a Microsoft C program. Section 3.3.11.1 gives an overview of Microsoft C error messages.

Several **CL** options are available to control the types of warnings generated at compile time, help with syntax checking, and verify compatibility between the actual arguments and formal parameters of a function during the early stages of program development. Sections 3.3.11.2–3.3.11.4 describe these options.

3.3.11.1 Understanding Error Messages

Error messages can appear at different stages of program development:

- In the compiling stage, the compiler generates a broad range of error and warning messages to help you locate errors and potential problems in your source files.
- During the linking stage, the linker is responsible for generating error messages.
- During program execution, any error messages you see are run-time error messages. This category includes messages about floating-point exceptions, which are errors generated by an 8087 or 80287 coprocessor.

Other utilities included in this package, such as the Microsoft Overlay Linker (**LINK**), the **MAKE** program-maintenance utility, and the **LIB** library manager, generate their own error messages. See the Microsoft CodeView and Utilities manual for a complete list of utility error messages.

When you are compiling and linking using the **CL** command, you may see both compiler and linker messages. The **LINK** program banner appears on the screen when the linking process begins. Compiler messages, if any, appear before the **LINK** banner, and linker messages, if any, appear after the banner. Compiler messages have numbers preceded by the letter C, and linker messages have numbers preceded by the letter L.

You can also distinguish the type of a message by its format. See Appendix E of this manual for a description of compiler error-message formats, a list of actual compiler error messages, and explanations of the circumstances that cause them. See Section C.2 of the Microsoft CodeView and Utilities manual for information about linker error messages.

Compiler error messages are sent to the standard output, which is usually your terminal. You can redirect the messages to a file or printer by using one of the MS-DOS redirection symbols: **>** or **>>**. Error redirection is especially useful in batch-file processing.

■ Example

Assume the following source file named **RM.C**:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char argv[];

    {
        register int i;
        char *name;

        for (i = 1; i < argc; ++i)
            if (unlink(name = argv[i])) {
                printf("couldn't delete %s : ", name);
                perror("");
            }
    }
```

The following command line redirects error messages to a file named **RM.ERR**:

```
CL RM.C > RM.ERR
```

In the command above, only output that ordinarily goes to the console screen is redirected. The error-message file **RM.ERR** contains the following information:

```
rm.c(11) : error C2065: 'arg' : undefined
rm.c(12) : warning C4047: '=' : different levels of indirection
```

Based on the errors generated, you can correct RM.C as shown below:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];           /* corrects warning C4047 */
{
    register int i;
    char *name;

    for (i = 1; i < argc; ++i) /* corrects error C2065 */
        if (unlink(name = argv[i])) {
            printf("couldn't delete %s : ", name);
            perror("");
        }
}
```

3.3.11.2 Setting the Warning Level (/W, /w)

■ Option

```
/W{0|1|2|3}
/w
```

You can suppress warning messages produced by the compiler by using the **/W** (for “warning”) option. Compiler warning messages are any messages beginning with C4; see Appendix E, “Error Messages,” for a full listing of these messages. Warnings indicate potential problems (rather than actual errors) with statements that may not be compiled as you intend. The **/W** options affect only source files given on the command line; they do not apply to object files.

The **/W0** option turns off warning messages. This option is useful when you compile programs that deliberately include questionable statements. The **/W0** option applies to the remainder of the command line or until the next occurrence of a **/W** option on the command line. The **/w** option has the same effect as the **/W0** option.

The **/W1** option (the default) causes the compiler to display most warning messages.

The **/W2** option causes the compiler to display an intermediate level of warning messages. Level-2 warnings may or may not indicate serious problems; they include warnings such as the following:

- Use of functions with no declared return type
- Failure to put **return** statements in functions with non-**void** return types
- Data conversions that would cause loss of data or precision

The **/W3** option displays the highest level of warning messages, including warnings about the uses of non-ANSI features and extended keywords and about function calls before the appearance of function prototypes in the program.

Note that the warning messages in Appendix E, “Error Messages,” indicate the warning level that must be set (that is, the number for the appropriate **/W** option) for the message to appear.

■ Example

```
CL /W3 CRUNCH.C PRINT.C
```

This example enables all possible warning messages when the **CRUNCH.C** and **PRINT.C** source files are compiled.

3.3.11.3 Checking Syntax (**/Zs**)

■ Option

/Zs

The **/Zs** option causes the compiler to perform only a syntax check on the source files that follow the option on the command line. This option provides a quick way to find and correct syntax errors before you try to compile and link a source file.

When you give the **/Zs** option, the compiler does not generate code or produce object files, object listings, or executable files. However, the compiler does display error messages if the source file has syntax errors. You can specify the **/Fs** option on the same command line to generate a source listing that shows these error messages. See Section 3.3.8.1 for more information about the **/Fs** option.

■ Example

```
CL /Zs TEST*.C
```

This command causes the compiler to perform a syntax check on all source files in the current working directory that begin with TEST and end with the default extension (.C). The compiler displays messages for any errors found.

3.3.11.4 Generating Function Declarations (/Zg)

■ Option

/Zg

The **/Zg** option generates a function declaration for each function defined in the source file. The function declaration includes the function return type and an argument-type list created from the types of the formal parameters of the function. Any function declarations already present in the source file are ignored.

The generated list of declarations is written to the standard output. It can be saved in a file using MS-DOS redirection.

When the **/Zg** option is used, the source file is not compiled. As a result, no object file or listing is produced.

The list of declarations is helpful for verifying that actual arguments and formal parameters of a function are compatible. You can save the list and include it in your source file to cause the compiler to perform type checking. The presence of a declared argument-type list for a function “turns on” the compiler’s type checking between actual arguments to a function (given in the function call) and the formal parameters of a function.

This type checking can be a helpful feature in writing and debugging C programs, especially when working with older C programs. Argument type checking is a recent addition to the C language, so many existing C programs will not have argument-type lists. See Chapters 4 and 7 of the *Microsoft C Language Reference* for more information about function declarations and argument-type lists.

You can use the **/Zg** option even if your source program already contains some function declarations. The compiler accepts more than one occurrence of a function declaration, as long as the declarations do not conflict. No conflict occurs when one declaration has an argument-type list and another declaration of the same function does not, as long as the return types are identical.

Note

If you use the **/Zg** option and your program contains formal parameters that have structure, enumeration, or union type (or pointers to such types), then the declaration for each structure, enumeration, or union type must have a tag. For example, use the following form:

```
struct tagA {
    .
    .
    .
} A;
```

Your program can include calls to Microsoft C run-time library routines. The include files provided with the Microsoft C run-time library contain function declarations that enable type checking on library calls.

■ **Example**

```
CL /Zg FILE.C > FILEDECLS.H
```

The example above causes the compiler to generate argument-type lists for functions defined in `FILE.C`. The list of declarations is redirected to `FILEDECLS.H`.

3.3.12 Preparing for Debugging (**/Zi**, **/Zd**, **/Od**)

■ **Options**

/Zi Creates object file for use with Microsoft CodeView debugger
/Zd Creates object file for use with Microsoft SYMDEB symbolic debug utility
/Od Disables code optimization to help with debugging

The **/Zi** option produces an object file containing full symbolic-debugging information for use with the CodeView debugger. This object file includes full symbol-table information and line numbers. If the **/Zi** option is given with no explicit **/O** options, all optimizations involving code motion and rearrangement are suppressed, although simple optimizations are still performed. If any explicit **/O** options are given, *all* requested optimizations are performed.

The **/Zd** option produces an object file containing line-number records corresponding to the line numbers of the source file. The **/Zd** option is useful when you want to pass an object file to the **SYMDEB** debugger, available with other Microsoft products. The debugger can use the line numbers to refer to program locations; however, only global symbol-table information is available with this product.

The **/Od** option tells the compiler not to perform most optimizations. Some peephole optimizations and other simple optimizations are still performed. (Without the **/Od** option, the default is to optimize.) You may want to use this option when you plan to use a symbolic debugger with your object file, since optimization can involve rearrangement of instructions that make it difficult for you to recognize and correct your code when debugging. However, turning off optimizations may increase the size of the code generated to the point where it might not be possible to link your program.

Other optimization options are discussed in Section 3.3.13, "Optimizing."

■ Example

```
CL /Zi /Od TEST.C
```

This command produces an object file named **TEST.OBJ** that contains line numbers corresponding to the line numbers of **TEST.C**. A source-listing file, **TEST.LST**, is also created. Limited optimization is performed.

3.3.13 Optimizing

The optimizing capabilities available with the Microsoft C Optimizing Compiler can reduce the storage space or execution time required for a program. This is achieved by eliminating unnecessary instructions and rearranging code. The compiler performs some optimizations by default. You can use the **/O** options, the **loop_opt** pragma (described in Section 3.3.13.1 under "Loop Optimization"), and the **intrinsic** pragma (described in Section 3.3.13.1 under "Generating Intrinsic Functions") to exercise greater control over the optimizations performed. In addition, you can use the **/Gs** option or **check_stack** pragma to reduce program size and speed up execution.

3.3.13.1 Controlling Optimization (/O Options)

■ Option

```
/Ostring
# pragma loop_ opt([ { on|off} ])
# pragma intrinsic(function1[,function2]...)
# pragma function(function1[,function2]...)
```

The /O options give you control over the optimization procedures that the compiler performs. One or more of the letters in *string* following the /O let you choose how the compiler performs optimization:

Letter	Optimizing Procedure
a	Relaxes alias checking
d	Disables optimization
i	Enables intrinsic functions
l	Enables loop optimization
p	Improves consistency of floating-point results
s	Favors code size during optimization
t	Favors execution speed during optimization (the default)
x	Maximizes optimization

The letters can appear in any order; for example, /Oat and /Ota have the same effect. More than one /O option can be given; the compiler uses the last /O option given if any conflict arises. Each option applies to all source files following that option on the command line.

The following sections discuss the various optimization options and their effects.

Relaxing Alias Checking (/Oa)

The **a** option letter can be used with the **l**, **s**, or **t** option letter to relax the assumptions the compiler makes about the use of “aliases” in the program. Aliases are multiple names (that is, symbolic references) for the same memory location in a program. Most commonly, aliases occur as a result of code similar to that shown below:

```
func ()
{
    int  x, *p;

    p = &x; /* now "x" and "*p" refer to the same */
           /* memory location                      */
    .
    .
    .
}
```

Use of the **/Oa** option can reduce the size of executable files and speed program execution. Its use is especially recommended when you also specify the **/Ol** option, since the compiler can detect a number of loop optimizations when the **/Oa** option is in effect that it cannot detect when **/Oa** is not in effect. However, before you specify **/Oa**, you must make sure that your program does not use aliases either directly or indirectly.

The use of aliases is important only if both names are actually used to reference the memory location. Otherwise the use is benign, and the **/Oa** option may be specified. The following example illustrates a benign use of aliases:

```
func ()
{
    int  x, *p;

    p = &x;

    .
    .
    .
    /* ...expressions involving only *p */
    .
    .
    .
}
```

Since all access to the memory location labeled **x** is through the pointer **p**, **x** has no significance in the function. To illustrate, **func** could be rewritten as the following pair of functions:


```
func1 ()
{
    int x;

    func2 (&x);
}

func2 (p)
int *p;
{
    .
    .
    .
    /* ...expressions involving *p */
    .
    .
    .
}
```

In this equivalent form, the alias created in `func1` is insignificant, since the memory location is not referenced at all and `func2` does not use aliases since `x` is not even in the scope of the function. The **/Oa** option can be safely specified in compiling either of these equivalent forms.

In addition to the obvious cases discussed above, aliases can be created through the use of pointers in other, more subtle ways. Two such cases involving the use of pointers as function arguments are illustrated below:

```
int x;

func (p)
int *p;
{
    .
    .
    .
    /* ...expressions involving *p and x */
    .
    .
    .
}
```

In the example above, `x` is a communal variable, so the function can be called with `func (&x)`. The **/Oa** option can be used safely only if it is known that `func` is never invoked with the address of `x` as an argument.

```

func(p1, p2)
int *p1, *p2;
{
    .
    .
    .
    /* ...expressions involving *p1 and *p2 */
    .
    .
    .
}

```

In the example above, the function may be invoked with the same value for both arguments (that is, `func(p,p)` or `func(&x,&x)`). Thus, the `/Oa` option can be safely specified only if it is known that the function is always called with distinct values for the two arguments.

One use of aliases occurs so frequently that a special provision has been made for it. When the compiler encounters a call to a function with address-type arguments, it always assumes that all variables whose addresses are passed to the function are modified. If such function calls appear in a program, the `/Oa` option can be specified safely even though the function call results in an alias for each variable whose address is passed. The example below illustrates how the compiler handles this case:

```

func1()
{
    int x, y, a, b;
    .
    .
    .
    x = a + b;

func2(&a);

    y = a + b;
}

```

In the example above, when the compiler encounters the function call `func2(&a)`, it assumes that the function modifies `a`, even if the `/Oa` option has been specified. The compiler generates code to evaluate each instance of the expression `a + b` rather than eliminating a common subexpression incorrectly.

Although you should convert programs that use aliases if you plan to compile them with the `/Oa` option, it is helpful to know the units of a program where the optimizations affected by the use of `/Oa` are applied. This information indicates where the uses of aliases are most likely to cause incorrect optimizations if `/Oa` is specified. The following list describes the program units where such optimizations are performed:

- All of the C optimizations, except for loop optimizations, that may be affected by the incorrect use of **/Oa** are applied at the level of basic blocks. In the Microsoft C Optimizing Compiler, the **/Oa** option can generally be used even if aliases are used, provided no memory location is referenced by more than one name within any basic block. (A “basic block” is a contiguous sequence of statements, with a unique entry point and exit point and no branching in between. In C programs, basic blocks most often appear as the clauses of **if** statements, **switch** statements, loop bodies, or function bodies, although they may also occur as sequences of statements delimited by user labels.)
- Loop optimizations are applied at the level of whole loop bodies. Thus, if loop optimization is enabled, **/Oa** can generally be used even if aliases are used, provided that no memory location is referenced by more than one name within any basic block or loop body.

Disabling Optimization (**/Od**)

The **/Od** option turns off most optimizations. This option is useful in the early stages of program development to avoid optimizing code that will later be changed. Because optimization may involve rearrangement of instructions, you may also want to specify the **/Od** option when you use a debugger with your program or when you want to examine an object-file listing. If you optimize before debugging, it can be difficult to recognize and correct your code. However, note that turning off or restricting optimization of a program usually increases the size of the generated code. If your program contains a module that is close to the 64K limit on compiled code, turning off optimization may cause the module to exceed the limit.

Generating Intrinsic Functions (**/Oi**)

The **/Oi** option tells the compiler to generate intrinsic functions instead of function calls for certain functions. Intrinsic functions may be in-line functions, may use special argument-passing conventions, or (in some cases) may do nothing. Programs that use intrinsic functions are faster because they do not include the overhead associated with function calls. However, they may be larger because of the additional code that is generated.

The following functions have intrinsic forms:

- **memset**, **memcpy**, and **memcmp**
- **strset**, **strcpy**, **strcmp**, and **strcat**
- **inp** and **outp**
- **_rotr**, **_rotr**, **_lrotr**, and **_lrotr**
- **min**, **max**, and **abs**

- **pow**, **log**, **log10**, and **exp**
- **sin**, **cos**, and **tan**
- **asin**, **acos**, **atan**, and **atan2**
- **sinh**, **cosh**, and **tanh**
- **sqrt**
- **floor**, **ceil**, **fabs**, and **fmod**

Note

Intrinsic versions of the **memset**, **memcpy**, and **memcmp** functions in compact- and large-model programs cannot handle huge arrays or huge pointers. To use huge arrays or huge pointers with these functions, you must compile your program with the huge memory model (that is, using the **/AH** option on the command line).

You can use the **intrinsic** pragma to generate intrinsic functions only for selected functions. This pragma has the following format:

```
# pragma intrinsic (function1 [,function2]...)
```

The **intrinsic** pragma affects the specified functions from the point where the pragma appears until either the end of the source file or the next **function** pragma specifying any of the same functions. The **function** pragma has the following format:

```
# pragma function (function1 [,function2]...)
```

Note that you can also use the **function** pragma selectively to generate function calls instead of intrinsic functions when you compile a program with the **/Oi** option.

Loop Optimization (/Ol)

The **/Ol** option tells the compiler to perform loop optimizations. For best performance, the **/Ol** option should be specified along with the **a** option letter (**/Oal**), since the compiler can detect more loop optimizations when it relaxes its assumptions about the use of aliases.

You can use the **loop_opt** pragma to turn loop optimization on or off for selected functions. When you want to turn off loop optimization, put the

following line before the code on which you don't want to perform loop optimization:

```
# pragma loop_opt (off)
```

Note that the preceding line disables loop optimization for all code that follows it in the source file, not just the routines on the same line. To reinstate loop optimization, insert the following line:

```
# pragma loop_opt (on)
```

If no argument is given to the **loop_opt** pragma, loop optimization reverts to the behavior specified on the command line: enabled if the **/Ox** or **/Ol** option is in effect, and disabled otherwise. The interaction of the **loop_opt** pragma with the **/Ol** and **/Ox** options is explained in greater detail in Table 3.4.

Table 3.4
Using the loop_opt Pragma

Syntax	Compiled with /Ox or /Ol?	Action
# pragma loop_opt()	no	Turns off optimization for loops that follow
# pragma loop_opt()	yes	Turns on optimization for loops that follow
# pragma loop_opt (on)	yes or no	Turns on optimization for loops that follow
# pragma loop_opt (off)	yes or no	Turns off optimization for loops that follow

Achieving Consistent Floating-Point Results (/Op)

The **/Op** option is useful when floating-point results must be consistent within a program. This option changes the way in which the program handles floating-point values by default.

Ordinarily the compiler stores each floating-point value in an 80-bit register. In subsequent references to that value, the compiler reads the value from the register. When the final value is written to memory, it is truncated, since floating-point types are allocated fewer than 80 bits of storage (32 bits for the **float** type and 64 bits for the **double** type). Thus, the value stored in the register may actually be more precise than the same value stored in a floating-point variable. Since the value is truncated each

time it is written to memory, over the course of the program the value stored in the machine register may become quite different from the value that is written to memory.

If you use the **/Op** option, when floating-point values are referenced the compiler reloads them from floating-point variables rather than from registers. Using **/Op** gives less precise results than using registers, and it may increase the size of the generated code. However, it gives you more control over the truncation (and hence the consistency) of floating-point values.

Optimizing for Speed and Code Size (**/Ot**, **/Os**)

When you do not give an **/O** option to the **CL** command, it automatically uses **/Ot**, meaning that program execution speed is favored in the optimization. Wherever the compiler has a choice between producing smaller (but perhaps slower) and larger (but perhaps faster) code, the compiler generates faster code. For example, when the **/Ot** option is in effect, the compiler generates intrinsic functions to perform shift operations on long operands.

To cause the compiler to favor smaller code size instead, use the **/Os** option. For example, when the **/Os** option is in effect, the compiler uses function calls to perform shift operations on long operands.

Producing Maximum Optimization (**/Ox**)

The **/Ox** option is a shorthand way to combine optimizing options to produce the fastest possible program. Its effect is the same as using the following options on the same command line:

```
/Oa /i /t /Gs
```

That is, the **/Ox** option relaxes alias checking; generates all intrinsics for the functions listed under "Generating Intrinsic Functions" above; performs loop optimizations; favors execution time over code size; and removes stack probes. Note that the interactions between the **/Ox** option and the **loop_opt** pragma are the same as those described in Table 3.4. See Section 3.3.13.2 for more information about stack probes and ways of controlling their use.

■ Examples

```
CL /Oa FILE.C
```


This command tells the compiler to perform loop optimizations and relax alias checking when it compiles `FILE.C`. The compiler favors program speed over program size, since the `/Ot` option is also specified by default.

```
CL /c /Os FILE.C
```

The command above favors code size over execution speed when `FILE.C` is compiled.

```
CL /Od *.C
```

The command above compiles and links all C source files with the default extension (`.C`) in the current directory and disables optimization. This command is most useful during the early stages of program development, since it improves compilation speed.

3.3.13.2 Removing Stack Probes (/Gs)

■ Options

/Gs

```
#pragma check_stack([ { on|off} ])
```

You can reduce the size of a program and speed up execution slightly by removing stack probes. You can do this either with the `/Gs` option or with the `check_stack` pragma.

A “stack probe” is a short routine called on entry to a function to verify that there is enough room in the program stack to allocate local variables required by the function. The stack probe routine is called at every function entry point. Ordinarily, the stack probe routine generates a stack overflow message when it determines that the required stack space is not available. When stack checking is turned off, the stack probe routine is not called, and stack overflow can occur without being diagnosed (that is, no error message is printed).

Use the `/Gs` option when you want to turn off stack checking for an entire module if you know that the program does not exceed the available stack space. For example, stack probes may not be needed for programs that make very few function calls, or that have only modest local variable requirements. In the absence of the `/Gs` option, stack checking is on.

Use the `check_stack` pragma when you want to turn stack checking on or off only for selected routines, leaving the default (as determined by the presence or absence of the `/Gs` option) for the rest. When you want to

turn off stack checking, put the following line before the definition of the function you don't want to check:

pragma check_stack (off)

Note that the preceding line disables stack checking for all routines that follow it in the source file, not just the routines on the same line. To reinstate stack checking, insert the following line:

pragma check_stack (on)

Note

For earlier versions of Microsoft C, the **check_stack** pragma had a different format: **check_stack+** to enable stack checking and **check_stack-** to disable stack checking. Although the Microsoft C Optimizing Compiler still accepts this format, its use is discouraged, since it may not be supported in future versions.

If no argument is given for the **check_stack** pragma, stack checking reverts to the behavior specified on the command line: disabled if the **/Gs** option is given, or enabled if otherwise. The interaction of the **check_stack** pragma with the **/Gs** option is explained in greater detail in Table 3.5.

Table 3.5

Using the check_stack Pragma

Syntax	Compiled with /Gs Option?	Action
# pragma check_stack()	yes	Turns off stack checking for routines that follow
# pragma check_stack()	no	Turns on stack checking for routines that follow
# pragma check_stack(on)	yes or no	Turns on stack checking for routines that follow
# pragma check_stack(off)	yes or no	Turns off stack checking for routines that follow

Note

The **/Gs** option should be used with great care. Although it can make programs smaller and faster, it may mean that the program will not be able to detect certain execution errors.

■ Example

```
CL /Oals /Gs FILE.C
```

This example optimizes the file `FILE.C` by removing stack probes with the **/Gs** option. The letters specified with the **/O** option tell the compiler to relax alias checking (**a**), perform loop optimization (**l**), and favor code size over program speed (**s**). If you want stack checking for only a few functions in `FILE.C`, you can use the **check_stack** pragma around the definitions of functions you want to check. Similarly, if you want to perform loop optimization on only a few functions in `FILE.C`, you can use the **loop_opt** pragma around the definitions of functions on which you want to perform loop optimization.

3.3.14 Enabling and Disabling Language Extensions (**/Ze**, **/Za**)

■ Option

/Ze Enables language extensions (default)
/Za Disables language extensions

The Microsoft C Optimizing Compiler is moving to support the the ANSI C standard. In addition, it offers a number of features beyond the features specified in the the ANSI C standard. These features are enabled when the **/Ze** (default) option is in effect and disabled when the **/Za** option is in effect. They include the following:

- The **cdecl**, **far**, **fortran**, **huge**, **near**, and **pascal** keywords
- Use of casts to produce lvalues, as in the following example:

```
int *p;  
((long *)p)++;
```

The preceding example could be rewritten to conform with the ANSI C standard as shown below:

```
p = (int *) ((char *)p + sizeof(long));
```

- Redefinitions of **extern** items as **static**, as in the example below:

```
extern int foo();
static int foo()
{ }
```

- Use of trailing commas (,) rather than an ellipsis (...) in function declarations to indicate variable-length argument lists, as in the following example:

```
int printf(char *, );
```

- Benign **typedef** redefinitions within the same scope, as in the following example:

```
typedef int INT;
typedef int INT;
```

- Use of mixed character and string constants in an initializer, as in the following example:

```
char arr[5] = {'a', 'b', "cde"};
```

- Use of bit fields with base types other than **unsigned int** or **signed int**

Use the **/Za** option if you will be porting your program to other environments. The **/Za** option tells the compiler to treat extended keywords as simple identifiers and disable the other extensions listed above. When you specify **/Za**, the compiler automatically defines the identifier **NO_EXT_KEYS**. In the include files provided with the Microsoft C Optimizing Compiler run-time library, this identifier is used with **#ifndef** to control use of the **cdecl** keyword on library function prototypes. For an example of this conditional compilation, see the file **stdio.h**.

3.3.15 Packing Structure Members (/Zp)

■ Option

```
/Zp[{ 1|2|4} ]
#pragma pack([ { 1|2|4} ])
```

When storage is allocated for structures, structure members are ordinarily stored as follows:

- Items of type **char** or **unsigned char**, or arrays containing items of these types, are byte aligned.
- Structures are word aligned; structures of odd size are padded to an even number of bytes.
- All other types of structure members are word aligned.

To conserve space, or to conform to existing data structures, you may want to store structures more or less compactly. The **/Zp** option and the **pack** pragma control how structure data are “packed” into memory.

Use the **/Zp** option when you want to specify the same packing for all structures in a module. When you give the **/Zp[n]** option, where *n* is 1, 2, or 4, each structure member after the first is stored on *n*-byte boundaries, depending on the option you choose. If you use the **/Zp** option without an argument, structure members are packed on 1-byte boundaries.

On some processors, the **/Zp** option may result in slower program execution because of the time required to unpack structure members when they are accessed. For example, on an 8086 processor, this option can reduce efficiency if members with **int** or **long** type are packed in such a way that they begin on odd-byte boundaries.

Use the **pack** pragma when you want to specify packing other than the packing specified on the command line for particular structures. Give the **pack(n)** pragma, where *n* is 1, 2, or 4, before structures that you want to pack differently. To reinstate the packing given on the command line, give the **pack()** pragma with no arguments.

Table 3.6 shows the interaction of the **/Zp** option with the **pack** pragma.

Table 3.6
Using the pack Pragma

Syntax	Compiled with /Zp Option?	Action
# pragma pack()	yes	Reverts to packing specified on the command line for structures that follow
# pragma pack()	no	Reverts to default packing for structures that follow
# pragma pack(<i>n</i>)	yes or no	Packs the following structures to the given byte boundary until changed or disabled

■ Example

CL /Zp PROG.C

This command causes all structures in the program PROG.C to be stored without extra space for alignment of members on **int** boundaries.

3.3.16 Setting the Stack Size (/F)

■ Option

/F hexnum

The */F* option sets the size of the program stack. A space must separate the */F* and *hexnum*.

The *hexnum* is a hexadecimal value representing the stack size in bytes. The value must be less than 0xFFFF hexadecimal (65,535 decimal).

If you do not specify this option, the start-up routine in the standard C library sets the default stack size to 2K.

If you get a stack-overflow message, you may need to increase the size of the stack. In contrast, if your program uses the stack very little, you may save some space by decreasing the stack size.

Note

You can also use the **EXEMOD** utility, described in Chapter 15 of the Microsoft CodeView and Utilities manual, to change the default stack size for C program files by modifying the executable-file header. The format of the executable-file header is discussed in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on MS-DOS.

The */F* option is a linking option that affects executable files only; it does not have any effect on source or object files.

Using the */F* option with the **CL** command has the same effect as using the */STACK* option with the **LINK** program. See Section 4.4 for more information about the */STACK* option.

■ Example

```
CL /F C00 *.OBJ
```

This example sets the stack size to C00 hexadecimal (3K decimal) for the program created by linking all of the object files in the current working directory.

3.3.17 Restricting the Length of External Names (/H)

■ Option

/H number

The **CL** command allows you to restrict the length of external (public) names by using the */H* option. The *number* is an integer specifying the maximum number of significant characters in external names. The space between */H* and *number* is optional.

When you use the */H* option, the compiler considers only the first *number* characters of external names used in the program. The program may contain external names longer than *number* characters; the extra characters are simply ignored.

The */H* option is typically used to conserve space or to aid in creating portable programs. The Microsoft C Optimizing Compiler imposes no restrictions on the length of external names (although it uses only the first 31 characters), but other compilers or linkers may produce errors when they encounter names longer than a predetermined limit.

3.3.18 Labeling the Object File (/V)

■ Option

/V string

Use the */V* (for “version”) option to embed a text string in an object file. The *string* must be enclosed in double quotation marks (") if it contains white-space characters or embedded double quotation marks. A backslash (\) must precede any embedded double quotation marks.

Object files are machine readable but are not easily read and understood by humans. A typical use of the */V* option is to label an object file with a version number or copyright notice.

■ Example

```
CL /V"Microsoft C Optimizing Compiler Version 5.0" MAIN.C
```

The above command places the string

```
Microsoft C Optimizing Compiler Version 5.0
```

in the object file MAIN.OBJ.

3.3.19 Suppressing Default-Library Selection (/Zl)

■ Option

/Zl

Ordinarily the compiler places the name of the default combined library for the memory-model and floating-point options you have chosen (**mLIBCf.LIB**) in the object file for the linker to read. This allows the appropriate library to be linked with a program automatically.

The **/Zl** option tells the compiler not to place the default library name in the object file. As a result, the object file is slightly smaller.

The **/Zl** option is useful when you are building a library of routines. Every routine in the library need not contain the default-library information. Although the **/Zl** option saves only a small amount of space for a single object file, the total space saved is significant in a library containing many object modules. When you link a library of object modules created *with* the **/Zl** option and a C program file compiled *without* the **/Zl** option, the default-library information is supplied by the program file.

■ Example

```
CL ONE.C /Zl TWO.C
```

The example above creates the following two object files:

- An object file named ONE.OBJ that contains the name of the standard C combined library (**SLIBCE.LIB**)
- An object file named TWO.OBJ that contains no default-library information

When `ONE.OBJ` and `TWO.OBJ` are linked, the default-library information in `ONE.OBJ` causes the given library to be searched for any unresolved references in either `ONE.OBJ` or `TWO.OBJ`.

3.3.20 Changing the Default `char` Type (/J)

■ Option

/J

In Microsoft C, the `char` type is signed by default, so if a `char` value is widened to `int` type, the result is sign extended. You can change this default to `unsigned` with the /J option, causing the `char` type to be zero extended when widened to `int` type. However, if a `char` value is explicitly declared `signed`, the /J option does not affect it, and the value is sign extended when widened to `int` type.

When you specify /J, the compiler automatically defines the identifier `_CHAR_UNSIGNED`, which is used with `#ifndef` in the `limits.h` include file to define the range of the default `char` type.

3.3.21 Controlling Stack and Heap Allocation

You can change the model used to allocate heap space by linking your program with one of the `mVARSTCK.OBJ` object files (where *m* is the first letter of the library you choose). These files are the small-, medium-, compact-, and large-model versions of a routine that allows the memory allocation functions (`malloc`, `calloc`, `_expand`, `_fmalloc`, `_nmalloc`, and `realloc`) to allocate items in unused stack space if they run out of other memory. The large-model version can also be used for huge-model programs.

Programs compiled and linked under Microsoft C run with a fixed stack size (the default size is 2048 bytes). The stack resides above static data, and the heap uses whatever space is left above the stack. However, for some programs a fixed-stack model may not be ideal; a model where the stack and heap compete for space is more appropriate. Linking with the `mVARSTCK.OBJ` object files gives you such a model: when the heap runs out of memory, it tries to use available stack space until it runs into the top of the stack. When the allocated space in the stack is freed, it is once again made available to the stack. Note that the stack cannot grow beyond the last allocated heap item in the stack or, if there are no heap items in the stack, beyond the size it was given at link time. Note also that while the heap can employ unused stack space, the reverse is not true: the stack cannot employ unused heap space.

When you link your program with one of the *mVARSTCK.OBJ* files, you should be wary of suppressing stack checking with the pragma `#check_stack`, or the `/Gs` or `/Ox` option. This is because stack overflow can occur more easily in programs that use this option, possibly causing errors that would be difficult to detect. (See Section 3.3.13.2, "Removing Stack Probes," and the section titled "Maximum Optimization" in Section 3.3.13.1, for more information on suppression of stack checking.)

■ Example

```
CL TEST.C SVARSTCK
```

This command line compiles `TEST.C` and then links the resulting object module with `SVARSTCK.OBJ`, the variable-stack object file for small-model programs.

3.3.22 Controlling the Calling Convention (/Gc)

■ Options

```
/Gc  
fortran  
pascal  
cdecl
```

The **fortran**, **pascal**, and **cdecl** keywords, and the `/Gc` option, allow you to control the function-calling and naming conventions so that your C programs can call and be called by functions that are written in FORTRAN and Pascal.

Because C, unlike other languages such as Microsoft Pascal and Microsoft FORTRAN, allows the user to write functions that take a variable number of arguments, it must handle function calls differently. Languages such as Pascal and FORTRAN normally push actual parameters to a function in left-to-right order, with the last argument in the list being the last one pushed on the stack. In contrast, C functions do not always know the number of actual parameters, so they must push their arguments from right to left, with the first argument in the list being the last one pushed.

Additionally, the calling function must remove the arguments from the stack in C (rather than having the called function do it, as in Pascal and FORTRAN). If the code for removing arguments is in the called function (as in Pascal and FORTRAN), it appears only once; if it is in the calling function (as in C), it appears every time there is a function call. Since

function calls are more numerous than individual functions, the Pascal/FORTRAN method is slightly smaller and more efficient.

The Microsoft C Optimizing Compiler has the ability to generate the Pascal/FORTRAN calling convention in one of several ways. The first is through the use of the **pascal** and **fortran** keywords. When these keywords are applied to functions, or to pointers to functions, they indicate a corresponding Pascal or FORTRAN function. Therefore, the correct calling convention must be used. In the following example, `sort` is declared as a function using the alternative calling convention:

```
short pascal sort(char *, char *);
```

The **pascal** and **fortran** keywords can be used interchangeably. Use them when you want to use the left-to-right calling sequence for selected functions only.

The second method for generating the Pascal/FORTRAN calling convention is to use the **/Gc** option. If you use the **/Gc** option, the entire module is compiled using the alternative calling convention. You might use this method to make it possible to call all the functions in a C module from another language, or to gain the performance and size improvement provided by this calling convention. When you use **/Gc** to compile a module, the compiler assumes that all functions called from that module use the Pascal/FORTRAN calling convention, even if the functions are defined outside that module. Thus, using **/Gc** would normally mean that you cannot call or define functions that take variable numbers of parameters, and that you cannot call functions such as the C library functions that use the C calling sequence. In addition, if you compile with the **/Gc** option, either you must declare the **main** function in the source program with the **cdecl** keyword, or you must change the start-up routine so that it uses the correct naming and calling conventions when calling **main**.

To overcome these restrictions, the **cdecl** keyword has been added to Microsoft C. This keyword is the “inverse” of the **fortran** and **pascal** keywords. When applied to a function or function pointer, it indicates that the associated function is to be called using the normal C calling convention. This allows you to write C programs which take advantage of the more efficient calling convention while still having access to the entire C library, other C objects, and even user-defined functions that can take variable-length argument lists.

For convenience, the **cdecl** keyword has already been applied to run-time library function declarations in the include files distributed with this compiler. Thus, the library functions can be referenced freely, no matter which calling conventions are used, as long as the include files containing the appropriate function declarations are included for each function that is referenced.

Use of the **pascal** and **fortran** keywords, or the **/Gc** option, also affects the naming convention for the associated item (or, in the case of **/Gc**, all items): the name is converted to uppercase (capital letters), and the leading underscore that C normally prefixes is not added. The **pascal** and **fortran** keywords can be applied to data items and pointers, as well as functions; when applied to data items or pointers, these keywords force the naming convention described above for that item or pointer.

The **pascal**, **fortran**, and **cdecl** keywords, like the **near**, **far**, and **huge** keywords, are disabled by use of the **/Za** option. If this option is given, these names are treated as ordinary identifiers, rather than keywords.

■ Examples

```
int cdecl var_print(char*,...);
```

In the example above, `var_print` is allowed to have a variable number of arguments by declaring it as a function using the normal right-to-left C function calling convention and naming conventions. The `cdecl` keyword overrides the left-to-right calling sequence set by use of the **/Gc** option when compiling the source file in which this declaration appears; if this file is compiled without the **/Gc** option, `cdecl` has no effect since it is the same as the default C convention.

For more information on mixed-language programming, see the *Microsoft Mixed-Language Programming Guide*.

```
float *pascal nroot(number, root)
```

The example above declares `nroot` to be a function returning a pointer to a value of type `float`. The function `nroot` uses the default calling sequence (left-to-right) and naming conventions for Microsoft FORTRAN and Pascal programs.

```
long pascal index
```

The example above simply changes the naming convention for the data item `index`: it is included in the object file in all capital letters and without a leading underscore.

3.3.23 Compiling for Windows Applications (/Aw, /Gw)

■ Options

/Aw
/Gw

The **/Aw** option controls the segment setup, and should be used for C programs that interface with the Microsoft Windows operating system. For more information, see Section 6.5.3, "Setting Up Segments."

You should use the **/Gw** option for developing applications to run in the Windows environment. See your *Microsoft Windows Software Development Kit* for details on how and when to use this option.

3.3.24 XENIX-Compatible Options

To provide as much compatibility as possible with XENIX C compilers, the **CL** command also accepts the options recognized by the **cc** command on XENIX systems. Many of these options are identical to the **CL** options given in this manual; others have identical functions but different names. The following options are identical in the MS-DOS and XENIX versions of C (except that a forward slash, /, is a valid option character in the MS-DOS version):

-c	-I <i>pathname</i>	-P
-C	-ND <i>name</i>	-V <i>string</i>
-D <i>name</i>	-NM <i>name</i>	-w
-E	-NT <i>name</i>	-W <i>number</i>
-EP	-O <i>letters</i>	-X
-F <i>number</i>		

Table 3.7 shows the XENIX options that do not map directly to the options accepted by the **CL** command.

Table 3.7

XENIX Options Accepted by the CL Command

XENIX Option	Task	CL Option
-dos	Performs cross-compilation to create MS-DOS-executable file	Same (meaningful only on XENIX)
-K	Removes stack probes from a program	/Gs
-L	Creates an object-listing file containing assembled object code and suppresses linking	/Fl /c
-Mstring	Sets the program configuration. The <i>string</i> may be any combination of s (small model), m (medium model), c (compact model), l (large model), h (huge model), e (enables far , near , huge , fortran , pascal , and cdecl keywords), 2 (enables 80286 code generation), b (reverses word order for items of type long), t[num] (sets data threshold for largest item in a segment), and d (compiles program so that stack segment not equal to data segment). The s , m , c , l , and h options are mutually exclusive.	-Me is equivalent to /Ze. -M2 is equivalent to /G2. -Mt[num] is equivalent to /Gt[num]. -Mb has no equivalent. -Ms is equivalent to /AS. -Mm is equivalent to /AM. -Mc is equivalent to /AC. -Md is equivalent to /Au. -Ml is equivalent to /AL. -Mh is equivalent to /AH.
-m name	Creates a map file	-Fmname
-nlnum	Sets the maximum length of external symbols	-Hnum
-o filename	Makes <i>filename</i> the name of the final executable program	/Feexefiles
-S	Creates an assembly source listing and suppresses linking	/Fa /c

3.4 Controlling Binary and Text Modes

Most C programs use one or more data files for input and output. Under MS-DOS, data files are ordinarily processed in “text” mode. In text mode, carriage-return–line-feed (CR-LF) combinations are translated into a single line-feed (LF) character on input. Line-feed characters are translated to CR-LF combinations on output.

In some cases you may want to process files without making these translations. In binary mode, CR-LF translations are suppressed.

Standard library routines such as **fopen** or **open** give you the option of overriding the default mode when you open a particular file. You can also change the default mode for an entire program from text to binary mode. Do this by linking your program with the file **BINMODE.OBJ**, which is supplied as part of your C compiler software. Simply add the path name of **BINMODE.OBJ** to the list of object file names when you link your program.

When you link with **BINMODE.OBJ**, all files opened in your program default to binary mode, with the exceptions of **stdin**, **stdout**, and **stderr**. However, linking with **BINMODE.OBJ** does not force you to process all data files in binary mode. You still have the option to override the default mode when you open the file.

Use the **setmode** library function when you want to change the default mode of **stdin**, **stdout**, or **stderr** from text to binary, or the default mode of **stdaux** or **stdprn** from binary to text. The **setmode** function can change the current mode for any file and is primarily used for changing the modes of **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**, which are not explicitly opened by users.

CHAPTER

4

LINKING WITH THE CL COMMAND

4.1	Introduction.....	115
4.2	The Default Linking Process.....	115
4.3	Passing Linker Information: The /link Option	115
4.3.1	Specifying Libraries	116
4.3.1.1	Linking with Additional Libraries.....	117
4.3.1.2	Looking in Different Locations for Libraries	117
4.3.1.3	Overriding Libraries Named in Object Files.....	117
4.3.2	Specifying Linker Options	119
4.3.2.1	Defining Linker Options on the CL Command Line	119
4.3.2.2	Defining Linker Options in the Environment.....	120
4.4	Linker Options.....	120

4.1 Introduction

Since the **CL** command controls linking as well as compiling, you can specify linker options and libraries other than the default combined library to be linked with your object files on the **CL** command line.

4.2 The Default Linking Process

When the **CL** command compiles a source file, it encodes the name of the appropriate library built by the **SETUP** program in the object file. The library name embedded in the library file is determined by the following:

- The memory-model (**/A**) option you give on the **CL** command line
- The floating-point (**/FP**) option you give on the **CL** command line

Table 3.1 shows the default library for each combination of memory-model and floating-point options. If you simply use the default memory-model option (**/AS**) or floating-point option (**/FPI**), **CL** encodes the name **SLIBCE.LIB**, the name of the standard library that corresponds to the defaults.

When an object file is linked, the linker looks for libraries matching the names encoded in the object file. The linker looks for these libraries first in the current working directory, then in any directory specified in the **LIB** environment variable. If it finds libraries matching these names, it automatically links those libraries with the object file.

The result is that you ordinarily do not need to give library names on the **CL** command line. See Section 4.3.1 for descriptions of the situations that require you to specify libraries to the **CL** command.

4.3 Passing Linker Information: The **/link** Option

To pass linker options or nondefault library names to the linker, give the following options on the **CL** command line after any source- and object-file names and **CL** options:

/link [*link-libinfo*]

Use the *link-libinfo* field to specify linker options, libraries, and library search paths. Note that library names can also be specified with source- and object-file names before the **/link** option on the command line, as long as the library names have the **.LIB** extension. These library names are searched before library names specified after the **/link** option. For more information

- See Section 4.4 for descriptions of the linker options that apply to Microsoft C.
- See Chapter 12 of the Microsoft CodeView and Utilities manual for complete descriptions of the available linker options.
- See Section 4.3.1 for information about specifying libraries and library search paths.

If you use the **/link** option with the **CL** command, it must be the last option on the command line.

Note

You cannot create an overlaid version of your program with the **CL** command; you must explicitly use the **LINK** command. See Section 12.5, "Using Overlays," of the Microsoft CodeView and Utilities manual for a description of overlays.

4.3.1 Specifying Libraries

To link object files with libraries other than the default library, give the names of the nondefault libraries on the **CL** command line. Library names appearing before **/link** must have the **.LIB** extension; library names appearing after **/link** may have blank extensions or no extensions. A space or plus sign (+) must follow each library name except the last.

Since the object file already contains the names of the correct combined library, you do not need to specify libraries unless you want to do any of the following:

- Link with additional libraries
- Look for libraries in different locations
- Override the use of the default library
- Link with object files compiled with Version 4.0 of Microsoft C
- Link with uncombined libraries provided with Version 5.0 of the Microsoft C Optimizing Compiler

4.3.1.1 Linking with Additional Libraries

If you specify additional libraries to **CL**, the linker searches the libraries you specify *before* it searches the default library to resolve external references in the object files. It searches the libraries you specify in their order of appearance on the command line.

If a library name includes a path specification, the linker searches only that path for the library.

If you specify only a library name (without a path specification), the linker searches in the following locations to find the given library file:

1. The current working directory
2. Any path specifications or drive names that you give in the *link-libinfo* field, in their order of appearance on the command line
3. The locations given by the **LIB** environment variable

If a library name without an extension appears after the **/link** option, the linker automatically supplies the **.LIB** extension. If you want to link a library file with an extension other than **.LIB**, you must specify the complete library name.

4.3.1.2 Looking in Different Locations for Libraries

You can tell the linker to look in different locations for libraries by giving a drive name or path specification in the *link-libinfo* field on the **CL** command line.

The linker looks for the default libraries in the same order as it looks for libraries given on the command line. See Section 4.3.1.1, “Linking with Additional Libraries,” for more information.

4.3.1.3 Overriding Libraries Named in Object Files

If you do not want to link with the library whose name is included in the object file, you can give the names of one or more different libraries instead. You might want to specify a different library name in the following cases:

- If you have renamed a standard library.
- If you want to link with a library for a different floating-point math package. Some restrictions apply; see Chapter 7, “Controlling Floating-Point Math Operations,” for more information.

- If you link with object files compiled with Version 3.0 or Version 4.0 of Microsoft C. In this case, the object files contain the names of the uncombined C libraries; you must override the default library names (see below) and explicitly specify the name of the combined Version 5.0 library or the uncombined libraries.
- If you want to link with uncombined Version 5.0 libraries. For example, you may not have used **SETUP** to build the appropriate library for a particular memory model, but may still want to link with the libraries for that memory model. In this case, you must specify uncombined libraries in the order shown below:
 1. The model-independent floating-point library **EM.LIB** (if you are using the emulator floating-point package) or **87.LIB** (if you are using the 8087/80287 floating-point package). You cannot link with **EM.LIB** or **87.LIB** if you have given the **/A** option on the **CL** command line.
 2. The model-dependent floating-point library **mLIBFP.LIB** or **mLIBFA.LIB** (where *m* indicates the memory model you are using).
 3. The model-dependent standard library **mLIBC.LIB** (where *m* indicates the memory model you are using).
 4. The model-independent code-helper library **LIBH.LIB**.

Note that you need to specify the uncombined libraries listed in steps 1 and 2 if you use floating-point math in your source program.

If you specify a new library name, the linker searches the new library to resolve external references before it searches the library specified in the object file.

If you want the linker to ignore the libraries named in the object file, you must use the **/NOD** linker option. This option tells the linker to ignore the default-library names encoded in the object files. Use this option with caution; see the discussion of the **/NOD** option in Section 4.4 for more information.

■ Example

```
CL FUN TEXT TABLE CARE /link C:\TESTLIB\ NEWLIBV3
```

This example links four object modules to create an executable file named **FUN.EXE**. The linker searches **NEWLIBV3.LIB** before searching the default libraries to resolve references. To locate **NEWLIBV3.LIB** and the default libraries, the linker searches the current working directory, then the **C:\TESTLIB** directory, and finally, the locations given by the **LIB** environment variable.

4.3.2 Specifying Linker Options

Linker options can be given explicitly on the **CL** command line, or they can be defined in the **CL** environment variable.

4.3.2.1 Defining Linker Options on the CL Command Line

When you use the **CL** command to invoke the linker, any linker options you specify (other than those supported by **CL** options such as **/F** and **/Fm**) must appear after the **/link** option on the command line. All options begin with the linker's option character, the forward slash (**/**).

The following sections outline the rules for specifying linker options on the **CL** command line.

Abbreviations

Since linker options are named according to their functions, some of these options are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique so that the linker can determine which option you want. (The minimum legal abbreviation for each option is indicated in the syntax of the option.)

For example, several options begin with the letters "NO"; therefore, abbreviations for those options must be longer than "NO" to be unique. You cannot use "NO" as an abbreviation for the **/NOIGNORECASE** option, since the linker cannot tell which of the options beginning with "NO" you intend. The shortest legal abbreviation for this option is **/NOI**.

Abbreviations must begin with the first letter of the option and must be continuous through the last letter typed. No gaps or transpositions are allowed.

Numerical Arguments

Some linker options take numerical arguments. A numerical argument can be any of the following:

- A decimal number from 0 to 65,535.
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with **0**. For example, the number 10 is a decimal number, but the number 010 is an octal number, equivalent to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with **0x** or **0X**. For example, 0x10 is a hexadecimal number, equivalent to 16 in decimal.

Differences from CL Options

If you are accustomed to using **CL** options, you should be aware that the linker options work in a slightly different manner. Keep the following differences in mind when you use linker options:

- Linker options can be abbreviated; **CL** options cannot. For example, the linker option **/NOIGNORECASE** can be abbreviated to **/NOI**.
- Case is not significant in linker options, as it is in **CL** options. For example, **/NOI** and **/noi** are equivalent.
- Linker options on the command line affect all files in the linking process, regardless of where the options appear in the *link-libinfo* field.

4.3.2.2 Defining Linker Options in the Environment

You can also define default linker options using the **CL** environment variable. Set the **CL** variable as shown below:

```
SET CL= ... /link option[ option]...
```

The options defined by **CL** are treated as if they appeared immediately after **/link** on the **CL** command line and before any linker options given on the command line.

Options defined in the environment must follow the rules outlined in Section 4.3.2.2.

4.4 Linker Options

This section summarizes the linker options that can be used with Microsoft C programs. Note that this section does not describe all available linker options; for a complete list, refer to Chapter 12 of the Microsoft CodeView and Utilities manual.

The following summary describes the linker options most commonly used with Microsoft C programs:

/HE[LP]

Causes the linker to display a list of the available options on the screen.

/P[AUSE]

Tells the linker to pause in the link session and display a message before it writes the executable (**.EXE**) file to disk.

/I[NFORMATION]

Displays information about the linking process, including the phase of linking and the names of the object files being linked.

This option is useful if you want to determine the locations of the object files being linked and the order in which they are linked.

/B[ATCH]

Tells the linker not to prompt you for a new path name whenever it cannot find a library or object file that it needs. When this option is used, the linker simply continues to execute without using the file in question. This option is intended primarily for users who employ batch or **MAKE** files to link many executable files with a single command and who do not want the linker to stop processing if it cannot find a required file.

/Q[UICKLIB]

Creates a Quick library for programs by the Microsoft QuickC Compiler. If you give this option, the linker creates a file with an extension of **.QLB** rather than an extension of **.EXE**. See Chapter 7 of the *Microsoft QuickC Compiler Programmer's Guide* for more information about creating Quick libraries.

/E[XEPACK]

Removes sequences of repeated bytes (typically null characters) and optimizes the load-time relocation table before creating the executable file. (The load-time relocation table is a table of references, relative to the start of the program, each of which changes when the executable image is loaded into memory and an actual address for the entry point is assigned.)

Executable files linked with this option may be smaller, and load faster, than files linked without this option. However, you cannot use the Symbolic Debug Utility (**SYMDEB**) or the CodeView window-oriented debugger to debug with packed files.

/NOD[EFAULTLIBRARYSEARCH]

Tells the linker *not* to search any library specified in the object file to resolve external references.

In general, C programs do not work correctly without the standard C libraries. Thus, if you use the **/NOD** option, you should explicitly specify the names of all required standard libraries.

/NOF[ARCALLTRANSLATION]
/F[ARCALLTRANSLATION]

Tells the linker whether or not to optimize intrasegment far calls. Used with the **/PACKCODE** option, the **/F** option can result in smaller executable files, reduced program-load time, and reduced execution time. The default is **/NOF**.

/NOP[ACKCODE]
/P[ACKCODE][:number]

Tells the linker whether or not to group contiguous logical code segments and assign each segment a base address that is the beginning of the group. **/NOP** is the default. The *number*, if given, specifies the limit at which to stop packing and start a new group. If the **/P** option is given with no *number*, 64K is the default.

/SE[GMMENTS]:number

Controls the number of segments that the linker allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1–1024 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. When you set the segment limit higher than 128, the linker allocates more space for segment information. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting *number* to reflect the actual number of segments in the program. The linker displays an error message if the number of segments allocated is too high for the amount of memory the linker has available.

/CP[ARMAXALLOC]:number

Sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory, where *number* is an integer in the range 1–65,535. The operating system uses this value when allocating space for the program before loading it. The Microsoft C start-up module cuts memory back to the larger of 64K or the amount of memory specified in this option; for programs with limited static data and heap usage, this option is unnecessary.

The following linker options can be used with Microsoft C programs, but they perform the same actions as **CL** options. Therefore, you do not need to use them unless you are compiling and linking in separate steps.

/M[AP][:*number*]

Creates a map file. This option is equivalent to using the **/Fm** option with the **CL** command, except that you can give a *number* argument with the **/M** option. The *number* argument is any positive integer (decimal, octal, or hexadecimal) up to 65,535 (decimal) specifying how many symbols are sorted in the map listing. If no *number* argument is given, a maximum of 2048 symbols is sorted. (In practice, the number of sorted symbols is limited by the amount of free heap space.) If a *number* argument is given, the alphabetical list of symbols does not appear in the map file.

/LI[NENUMBERS]

Creates a map file and includes the line numbers and associated addresses of the source program. This option is equivalent to using the **/Zd** option with the **CL** command. See Section 3.3.12 for more information about the **/Zd** option.

/ST[ACK]:*number*

Specifies the size of the stack for your program, where *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535 (decimal) representing the size, in bytes, of the stack. This option is equivalent to using the **/F** option of the **CL** command. See Section 3.3.16 for more information about the **/F** option.

/CO[DEVIEW]

Prepares for debugging with the CodeView window-oriented debugger provided with Version 5.0 of the Microsoft C Optimizing Compiler. This option is equivalent to using the **/Zi** option of the **CL** command. See Section 3.3.12 for more information about the **/Zi** option.

The following linker options can be used with Microsoft C programs, but they are never required, since they request actions that the **CL** command or the Microsoft C Optimizing Compiler performs automatically:

/NOI[GNORECASE]

Tells the linker to distinguish between uppercase and lowercase letters; for example, the linker would consider ABC, abc, and Abc to be three separate names. The **CL** command uses the **/NOI** option automatically; if you want to link without using **/NOI**, you must invoke the linker with the **LINK** command instead of using **CL**.

/DO[SSEG]

Forces segments to be ordered as follows:

1. All segments with a class name ending in **CODE**
2. All other segments outside **DGROUP** (that is, **FAR_DATA** and **BSS**)
3. **DGROUP** segments, in the following order:
 - a. Any segments of class **BEGDATA** (this class name is reserved for Microsoft use)
 - b. Any segments not of class **BEGDATA**, **BSS**, or **STACK**
 - c. Segments of class **BSS**
 - d. Segments of class **STACK**

C programs compiled with Version 5.0 of the Microsoft C Optimizing Compiler always use this segment order by default. See Section 6.7, "Naming Modules and Segments," for a discussion of the segment names used by the Microsoft C Optimizing Compiler.

CHAPTER

5

RUNNING C PROGRAMS ON MS-DOS

5.1	Introduction.....	127
5.2	Passing Command-Line Data to a Program.....	127
5.2.1	Expanding Wild-Card Arguments	130
5.2.2	Suppressing Command-Line Processing.....	131
5.3	Returning an Exit Code.....	131
5.4	Suppressing Null-Pointer Checks.....	132

5.1 Introduction

After compiling a program with the Microsoft C Optimizing Compiler and linking with the linker, you will have an executable file with the extension **.EXE** that can be run from the MS-DOS prompt.

MS-DOS uses the **PATH** environment variable to find executable files. You can execute your program from any directory, as long as the executable program file is either in your current working directory or in one of the directories on the path set in the **PATH** environment variable.

Your program can also be executed by other programs, or you can write it so that it will be capable of executing other programs or MS-DOS internal commands. The **spawn**, **exec**, and **system** routines provided in the run-time library allow your program to execute other programs and MS-DOS commands. See the *Microsoft C Run-Time Library Reference* for a description of these routines.

MS-DOS has several other unique capabilities that your program can use if you write the program to take advantage of them. Among these capabilities are the following:

- Receiving arguments from MS-DOS
- Reading information from the MS-DOS environment table
- Sending a message to MS-DOS by returning an exit code

This chapter explains how to write programs to take advantage of these features, and how to use them once your program is completed.

5.2 Passing Command-Line Data to a Program

Your C program can access data from a command line or from the environment table. You can use the MS-DOS **SET** or **PATH** command to place data in the environment table. See Section 2.4.5, "Setting Up the Environment," for a discussion of environment variables. Command-line data are arguments that appear on the same line as the program name when you execute the program.

To pass data to your program on the command line, give one or more arguments after the program name when you execute the program. Each argument must be separated from the arguments around it by one or more spaces or tab characters, and may be enclosed in quotation marks (" "). If you want to give a single argument that includes spaces or tab characters,

enclose the argument in quotation marks. For example, if your C program is called `TEST.EXE`, you might give it the following command line:

```
TEST 42 "de f" 16
```

In this case, the program will be executed and three arguments will be passed: 42, `de f`, and 16.

MS-DOS stores the command-line arguments in the MS-DOS program header. The C run-time library (which becomes part of your program during linking) in turn stores each argument from the program header as a null-terminated string in an array of strings. MS-DOS limits the combined length of all arguments on the command line (including the program name) to 128 bytes. If you provide a longer command line, additional characters are ignored.

For a C program to read the data from the command line, the program should declare two variables as arguments to the **main** function. These variables and their contents are listed in Table 5.1.

Table 5.1
Argument Variables

Variable	Contents
<i>argc</i>	Number of arguments passed
<i>argv</i>	Array of strings containing arguments

By declaring these variables as arguments to **main**, you make them available as local variables in the **main** function. The example below illustrates how to declare these arguments:

```
main (argc, argv)
int argc;
char *argv[ ];
```

The number of arguments appearing on the command line is passed as the integer variable *argc*, and the command line is passed to the program as the array of strings pointed to by *argv*.

The first argument of any command line is the name of the program to be executed. Therefore, the program name is the first string stored in *argv*, at *argv* [0]. Since a program name must be given to run the program, the integer value of *argc* is always at least 1. Therefore, if you pass two arguments to your program, *argc* will have a value of 3 (two arguments and the program name).

The first argument following the program name is stored at *argv* [1], the second is stored at *argv* [2], and so on, to the last argument.

Note

Under versions of MS-DOS earlier than 3.0, the program name normally stored in *argv* [0] is not available. References to *argv* [0] yield the string "C." Under MS-DOS versions 3.0 and later, references to *argv* [0] give the program name.

There is a third argument passed to the **main** function: *envp*, a pointer to the environment table. This argument is an extension provided by the Microsoft C Optimizing Compiler to support code ported from XENIX and other UNIX-like systems. When specified, it follows *argv* and is declared as shown below:

```
char *envp[ ];
```

Although you can use this pointer to access the value of environment settings, this usage is nonstandard and is not recommended. The **putenv** and **getenv** routines from the C run-time library accomplish the same task, and are easier and safer to use. Using the **putenv** routine may change the location of the environment table in memory, depending on memory requirements. Therefore, the value given to *envp* at the beginning of the program execution may not be valid throughout the program's execution. In contrast, the **putenv** and **getenv** routines access the environment table properly, even when its location changes. These routines use the global variable **environ** (described in the *Microsoft C Run-Time Library Reference*), which always points to the correct table location.

■ Example

```
MYPROG ABC "abc e" 3 8
```

This command line executes the program named MYPROG and passes the four command-line arguments to the **main** function. The arguments are stored as null-terminated strings, and the number of arguments is stored in *argc*. To access the last argument, for example, you would use an expression like the following:

```
argv[argc - 1]
```

Since the value of *argc* is 5 (counting the program name as an argument), this expression is equivalent to *argv* [4], or the fifth string of the array.

5.2.1 Expanding Wild-Card Arguments

You can use the MS-DOS wild-card characters, the question mark (?) and the asterisk (*), to specify file-name and path-name arguments on the command line. To prepare for using wild cards, you must link your object file with the **SETARGV.OBJ** object file.

This object file is included with your compiler software. If you don't link with this object file, your program does not expand wild-card characters on the command line, interpreting them instead as literal question marks and asterisks.

The **SETARGV.OBJ** file expands the wild-card characters in the same manner as MS-DOS. (See your DOS user's guide if you are unfamiliar with these characters.) Enclosing an argument in quotation marks (" ") suppresses the wild-card expansion. Within quoted arguments, you can represent quotation marks literally by preceding the double-quotation-mark character with a backslash (\), as shown below:

```
"*\\"argument\\"*
```

If no matches are found for the wild-card argument, the argument is passed literally. For example, if the argument `B:*.C` is given, but no files with the extension `.C` are found in the root directory of Drive B, the argument is passed as the string `B:*.C`.

If your programs frequently expand wild-card characters, you may want to put the wild-card routines (**SETARGV.OBJ**) in the appropriate standard C combined library (*mLIBCf.LIB*) so that they are linked with your program automatically. To do this, use the Microsoft Library Manager (**LIB**) to extract the module named `_setargv` from the library (the module name is the same in all four libraries) and insert **SETARGV**. When you replace `_setargv`, wild-card expansions are always performed on command-line arguments. **LIB** is described in Chapter 13 of the Microsoft CodeView and Utilities manual.

■ Example

```
CL BETA \LIB\SETARGV  
BETA *.INC "WHY?" \"HELLO\"
```

In this example, **SETARGV.OBJ**, which is in the directory `\LIB`, is linked with `BETA.OBJ`, producing the executable file `BETA.EXE`. When `BETA.EXE` is executed, the wild-card character `*` is expanded, causing all file names with the extension `.INC` in the current working directory to be passed as arguments to the `BETA` program. The second command-line argument, `WHY?`, is enclosed in quotation marks, so expansion of the wild-card character `?` is suppressed and the argument `WHY?` is passed literally.

In the third argument, the backslashes cause the quotation marks to be represented literally, so the argument "HELLO" (including the quotation marks) is passed.

5.2.2 Suppressing Command-Line Processing

If your program does not take command-line arguments, you can save a small amount of space by suppressing use of the library routine that performs command-line processing. This routine is called `_setargv`. To suppress its use, define a routine that does nothing in the same file that contains the `main` function, and name it `_setargv`. The call to `_setargv` will be satisfied by your definition of `_setargv`, and the library version will not be loaded.

Similarly, if you never access the environment table through the `envp` argument, you can provide your own empty routine to be used in place of `_setenvp`, the environment-processing routine.

If your program makes calls to the `spawn` or `exec` routines in the C runtime library, you should not suppress the environment-processing routine, since this routine is used to pass an environment from the parent process to the child process.

■ Example

```
_setargv ()
{
}

_setenvp ()
{
}
```

The example above shows how to define the `_setargv` and `_setenvp` functions to suppress command-line and environment processing. It is recommended that you place these definitions in the file containing the `main` function.

5.3 Returning an Exit Code

Your program can return an exit code (sometimes called a return code) as a means of leaving a message for MS-DOS. The exit code can then be used by MS-DOS batch files or other programs that test exit codes (for example, the **MAKE** program-maintenance utility). Exit codes and their uses are discussed in more detail in Appendix A, "Using Exit Codes."

Exit codes are returned through the **main** function. This function, like any other C function, can return a value. The value is of **int** type, and is passed to MS-DOS as the exit code of the executed program. This exit code can be checked with the **IF ERRORLEVEL** command in MS-DOS batch files. (See your DOS user's guide for more information about using batch files.)

To cause the **main** function to return a specific value to MS-DOS, you should use a **return** statement or the **exit** function to specify the value to be returned. For example, if the **main** function in a program terminates with either the statement `return (6);` or `exit (6);` the value 6 is returned to MS-DOS. If neither of these methods is used, the return code is undefined.

■ Example

```
#define TRUE    1
#define FALSE   0

int error = FALSE;

main()
{
    .
    .
    .
    if (error) return (1);
    else return (0);
}
```

In the example above, the value 1 would be returned if the variable `error` were set to `TRUE` somewhere within the body of the program. Otherwise, 0 would be returned to MS-DOS. The example program follows the convention of returning 0 if the program is successful, and some larger number if an error is encountered.

5.4 Suppressing Null-Pointer Checks

When you execute your C program, a special error-checking routine is automatically invoked after your program has terminated to determine whether the contents of the **NULL** segment have changed. If they have, the routine displays the following error message:

```
run-time error R6001
- null pointer assignment
```

The **NULL** segment is a special location in low memory that is normally not used. If the contents of the **NULL** segment change during a program's execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. Note that your program can contain null pointers without generating this message; the message appears only when you write to a memory location through the null pointer.

This error does not cause your program to terminate; the error is detected and the error message is printed following the normal termination of the program.

Note

The null-pointer error message reflects a potentially serious error in your program. Although a program that produces this error may appear to operate correctly, it is likely to cause problems in the future and may fail to run in a different operating environment.

The library routine that performs the null-pointer check is named **_nullcheck**. You can suppress the null-pointer check for a particular program by defining your own routine named **_nullcheck** that does nothing. The call to **_nullcheck** will be satisfied by your definition of **_nullcheck**, and the library version will not be loaded. It is recommended that you place the **_nullcheck** definition in the file containing the **main** function.

CHAPTER

6

WORKING WITH MEMORY MODELS

6.1	Introduction.....	137
6.2	Near, Far, and Huge Addressing	137
6.3	Using the Standard Memory Models	139
6.3.1	Creating Small-Model Programs.....	140
6.3.2	Creating Medium-Model Programs	141
6.3.3	Creating Compact-Model Programs.....	141
6.3.4	Creating Large-Model Programs.....	142
6.3.5	Creating Huge-Model Programs.....	143
6.4	Using the near, far, and huge Keywords	144
6.4.1	Library Support for near, far, and huge	145
6.4.2	Declaring Data with near, far, and huge	146
6.4.3	Declaring Functions with the near and far Keywords	148
6.4.4	Pointer Conversions	150
6.5	Creating Customized Memory Models.....	152
6.5.1	Code Pointers.....	153
6.5.2	Data Pointers	153
6.5.3	Setting Up Segments.....	154
6.5.4	Library Support for Customized Memory Models	155
6.6	Setting the Data Threshold.....	156
6.7	Naming Modules and Segments	157
6.8	Specifying Text and Data Segments.....	159

6.1 Introduction

You can gain greater control over how your program uses memory by specifying the memory model for the program. If you do not specify a memory model, **CL** uses the small memory model by default. The small memory model is sufficient for most programs.

You cannot use the small memory model if your program satisfies one or more of the following three conditions:

1. Your program has more than 64K of code.
2. Your program has more than 64K of data.
3. Your program contains individual arrays that need to be larger than 64K.

If you decide that the small memory model will not be adequate for your program, you have four options for larger memory models:

1. You can specify one of the other standard memory models (medium, compact, large, or huge) using one of the **/A** options.
2. You can create a mixed-model program using the **near**, **far**, and **huge** keywords.
3. You can create your own customized memory model using the **/Astring** option.
4. Method 2 can be combined with either method 1 or method 3.

6.2 Near, Far, and Huge Addressing

Understanding the terms “near,” “far,” and “huge” is crucial to understanding the concept of memory models. These terms indicate how data can be accessed in the segmented architecture of the 80x86 family of microprocessors (8086, 80186, 80286).

DOS loads the code and data allocated by your program into “segments” in physical memory. Each segment is up to 64K long. Since separate

segments are always allocated for the program code and data, the minimum number of segments allocated for a program is two; these two segments, required for every program, are called the default segments. The small memory model uses only the two default segments. The other memory models discussed in this chapter allow more than one code segment per program, more than one data segment per program, or both.

In the 80x86 family of microprocessors, all memory addresses consist of two parts:

1. A 16-bit number that represents the base address of a memory segment
2. Another 16-bit number that gives an offset within that segment

The architecture of the 80x86 microprocessor is such that code can be accessed within the default code or data segment using just the 16-bit offset value. This is possible because the segment addresses for the default segments are always known. This 16-bit offset value is called a “near” address, and can be accessed with a “near” pointer. Since only 16-bit arithmetic is required to access any near item, near references to code or data are smaller and more efficient.

When data or code lie outside the default segments, the address must use both the segment and offset values. Such addresses are called “far” addresses, and can be accessed by using “far” pointers in a C program. Accessing far data or code items is more expensive in terms of program speed and size, but using them allows your programs to address all memory, rather than just a 64K piece.

There is a third type of address in Microsoft C: the “huge” address. A huge address is similar to a far address in that both consist of a segment value and an offset value; but the two differ in the way address arithmetic is performed on pointers. Because items (both code and data) referenced by far pointers are still assumed to lie completely within the segment in which they start, pointer arithmetic is done only on the offset portion of the address. This gain in pointer arithmetic efficiency is achieved, however, by limiting the size of any single item to 64K. With data items, huge pointers overcome this size limitation: pointer arithmetic is performed on all 32 bits of the data item's address, thus allowing data items referenced

by huge pointers to span more than one segment, provided they conform to the rules outlined in Section 6.3.5, “Creating Huge-Model Programs.”

The rest of this chapter deals with the various methods you can use to control whether your program makes far, near, or huge calls to access code or data.

6.3 Using the Standard Memory Models

The libraries created by the **SETUP** program support five standard memory models. Using the standard memory models is the simplest way to control how your program accesses code and data in memory.

When you use the standard memory models, the compiler handles library support for you. The library corresponding to the memory model you specify is used automatically. Each memory model has its own library, except for the the huge memory model, which uses the large-model library.

The advantage of using standard models for your programs is simplicity. In the standard models, memory management is specified by compiler options; since the standard models do not require the use of extended keywords, they are the best way to write code that can be ported to other systems (particularly systems that do not use segmented architectures).

The disadvantage of using standard memory models exclusively is that they may not produce the most efficient code. For example, if you have an otherwise small-model program containing a large array that pushes the total data size for your program over the 64K limit for small model, it may be to your advantage to declare the one array with the **far** keyword, while keeping the rest of the program small model, as opposed to using the standard compact memory model for the entire program. For maximum flexibility and control over how your program uses memory, you can combine the standard-memory-model method with the **near**, **far**, and **huge** keywords described in Section 6.4.

The **/A** option for **CL** is used to specify one of the five standard memory models (small, medium, compact, large, or huge) at compile time. These options are discussed in the next five sections.

Note

In the following sections, which describe in detail the different memory-model addressing conventions, it is important to keep in mind two common features of all five models:

1. No *single* source module can generate 64K or more of code.
 2. No *single* data item can exceed 64K, unless it appears in a huge-model program or it has been declared with the **huge** keyword.
-

6.3.1 Creating Small-Model Programs

■ Option

/AS

The small-model option tells the compiler to create a program that occupies the two default segments: one for code and one for data.

Small-model programs are typically C programs that are short or have a limited purpose. Since code and data for these programs are each limited to 64K, the total size of a small-model program can never exceed 128K. Most programs fit easily into this model.

The default in small-model programs is that both code and data items are accessed with near addresses. You can override the default for data by using the **far** or **huge** keywords, and the default for code by using the **far** keyword (**huge** is relevant only to data items—specifically, arrays and pointers to arrays).

The compiler creates small-model programs by default when you do not specify a memory model. The **/AS** option is provided for completeness; you need never give it explicitly.

6.3.2 Creating Medium-Model Programs

■ Option

`/AM`

The medium-model option provides a single segment for program data, and multiple segments for program code. Each source module is given its own code segment.

Medium-model programs are typically C programs that have a large number of program statements (more than 64K of code), but a relatively small amount of data (less than 64K). Program code can occupy any amount of space and is given as many segments as needed; total program data cannot be greater than 64K. The medium model provides a useful trade-off between speed and space, since most programs refer more frequently to data items than to code.

6.3.3 Creating Compact-Model Programs

■ Option

`/AC`

The compact-model option directs the compiler to allow multiple segments for program data but only one segment for the program code.

Compact-model programs are typically C programs that have a large amount of data, but a relatively small number of program statements. Program data can occupy any amount of space and are given as many segments as needed.

The default in compact-model programs is that code items are accessed with near addresses and data items are accessed with far addresses. You can override the default by using the **near** and **huge** keywords for data, and the **far** keyword for code.

Note

Note that in medium and compact models, **NULL** must be used carefully in certain situations. **NULL** actually represents a null data pointer. In memory models where code and data pointers are the same size, it can be used with either. However, in memory models where code and data pointers are different sizes, this is not the case. Consider the following example:

```
void func1(char *dp)
{
    .
    .
    .
}

void func2(char (*fp)(void))
{
    .
    .
    .
}

main()
{
    func1(NULL);
    func2(NULL);
}
```

This example passes a 16-bit pointer to both `func1` and `func2` if compiled in medium model, and a 32-bit pointer to both `func1` and `func2` if compiled in compact model, unless prototypes are added to the beginning of the program to indicate the types, or an explicit cast is used on the argument to `func1` (compact model) or `func2` (medium model).

6.3.4 Creating Large-Model Programs

■ Option

`/AL`

The large-model option allows the compiler to create multiple segments as needed for both code and data.

Large-model programs are typically very large C programs that use a large amount of data storage during normal processing.

The default in large-model programs is that both code and data items are accessed with far addresses. You can override the default by using the **near** and **huge** keywords for data, and the **near** keyword for code.

6.3.5 Creating Huge-Model Programs

■ Option

/AH

The huge-model option is similar to the large-model option, except that the restriction on the size of individual data items is removed for arrays.

Some size restrictions apply to elements of huge arrays where the array is larger than 64K, however. To provide efficient addressing, array elements are not permitted to cross segment boundaries. This has the following implications:

1. No array element can be larger than 64K.
2. For any array larger than 128K, all elements must have a size in bytes equal to a power of 2 (that is, 2 bytes, 4 bytes, 8 bytes, 16 bytes, and so on). However, if the array is 128K or smaller, its elements may be any size, up to and including 64K.

In huge-model programs, care must be taken when using the **sizeof** operator or when subtracting pointers. The C language defines the value returned by the **sizeof** operator to be an **unsigned int** value, but the size in bytes of a huge array is an **unsigned long** value. To solve this discrepancy, the Microsoft C Optimizing Compiler produces the correct size of a huge array when a type cast like the following is used:

```
(unsigned long) sizeof (huge_item)
```

Similarly, the C language defines the result of subtracting two pointers as an **int** value. When subtracting two huge pointers, however, the result may be a **long int** value. The Microsoft C Optimizing Compiler gives the correct result when a type cast like the following is used:

```
(long) (huge_ptr1 - huge_ptr2)
```


6.4 Using the near, far, and huge Keywords

One limitation of the predefined memory-model structure is that, when you change memory models, all data and code address sizes are subject to change. However, the Microsoft C Optimizing Compiler lets you override the default addressing convention for a given memory model and access items with either a near, far, or huge pointer. This is done with the **near**, **far**, and **huge** keywords. These special type modifiers can be used with a standard memory model to overcome addressing limitations for particular data or code items, or to optimize access to these items, without changing the addressing conventions for the program as a whole. Table 6.1 explains how the use of these keywords affects the addressing of code or data, or pointers to code or data.

Table 6.1

Addressing of Code and Data Declared with near, far, and huge

Keyword	Data	Function	Pointer Arithmetic
near	Reside in default data segment; referenced with 16-bit addresses (pointers to data are 16 bits)	Assumed to be in current code segment; referenced with 16-bit addresses (pointers to functions are 16 bits)	Uses 16 bits
far	May be anywhere in memory, not assumed to reside in current data segment; referenced with 32-bit addresses (pointers to data are 32 bits)	Not assumed to be in current code segment; referenced with 32-bit address (pointers to functions are 32 bits)	Uses 16 bits
huge	May be anywhere in memory, not assumed to reside in current data segment; individual data items (arrays) can exceed 64K in size; referenced with 32-bit addresses (pointers to data are 32 bits)	Not applicable to code	Uses 32 bits for data

Note

The **near**, **far**, and **huge** keywords are not a standard part of the C language; they are meaningful only for systems that use a segmented architecture similar to that of the 80x86 microprocessors. Keep this in mind if you want your code to be ported to other systems.

In the Microsoft C Optimizing Compiler, the **near**, **far**, and **huge** keywords are enabled by default. To treat these keywords as ordinary identifiers, you must give the **/Za** option at compile time. This option is useful if you are concerned with porting C programs from environments in which these are not keywords; for instance, if you are porting a program in which one of these words is used as a label. See Section 3.3.14 for further information about the use and effects of the **/Za** option.

6.4.1 Library Support for near, far, and huge

When using the **near**, **far**, and **huge** keywords to modify addressing conventions for particular items, you can usually use one of the standard libraries (small, compact, medium, or large) with your program. The large-model libraries are also appropriate for use with huge-model programs. However, you must use care when calling library routines. In general, you cannot pass far pointers, or the addresses of far data items, to a small-model library routine. (Some exceptions to this statement are the library routines **malloc** and **free** and the **printf** family of functions.) Of course, you can always pass the *value* of a far item to a small-model library routine. For example:

```
long far time_val;

time(&time_val);           /* Illegal */
printf("%ld\n", time_val); /* Legal */
```

If you use the **near**, **far**, or **huge** keyword, it is strongly recommended that you use function prototypes with argument-type lists to ensure that all pointer arguments are passed to functions correctly. See Section 6.4.4, “Pointer Conversions,” for more information.

For more information on library routines and memory models, see Section 2.11, “Using Huge Arrays with Library Functions,” in the *Microsoft C Run-Time Library Reference*.

6.4.2 Declaring Data with **near**, **far**, and **huge**

The **near**, **far**, and **huge** keywords modify either objects or pointers to objects. When using them to declare data or code (or pointers to data or code), keep the following rules in mind:

- The keyword always modifies the object or pointer immediately to its right. In complex declarators, think of the **far** keyword and the item to its right as being a single unit. For example, in the case of the declarator

```
char far* *p;
```

p is a pointer (whose size depends on the specified memory model) to a far pointer to **char**. See the *Microsoft C Language Reference* for complete rules governing the use of special keywords in complex declarations.

- If the item immediately to the right of the keyword is an identifier, the keyword determines whether the item will be allocated in the default data segment (**near**) or a separate data segment (**far** or **huge**). For example,

```
char far a;
```

allocates *a* as an item of type **char** with a far address.

- If the item immediately to the right of the keyword is a pointer, the keyword determines whether the pointer will hold a near address (16 bits), a far address (32 bits), or a huge address (also 32 bits). For example,

```
char far *p;
```

allocates *p* as a far pointer (32 bits) to an item of type **char**.

■ Examples

The examples in this section show data declarations using the **near**, **far**, and **huge** keywords.

```
char a[3000];           /* small-model program */
char far b[30000];
```

The first declaration in the example above allocates the array *a* in the default data segment. By contrast, the array *b* in the second declaration may be allocated in any far data segment. Since these declarations appear in a small-model program, array *a* probably represents frequently used data that were deliberately placed in the default segment for fast access. Array *b* probably represents seldom used data that might make the

default data segment exceed 64K and force the programmer to use a larger memory model if the array were not declared with the **far** keyword. The second declaration uses a large array, because it is more likely that a programmer would want to specify the address allocation size for items of substantial size.

```
char a[3000];           /* large-model program */
char near b[3000];
```

In the example above, access speed would probably not be critical for array **a**. Even though it may or may not be allocated within the default data segment, it is always referenced with a 32-bit address. Array **b** is explicitly allocated **near** to improve speed of access in this memory model (large).

```
char huge a[70000];     /* small-model program */
char huge *pa;
```

In the small-model program above, **a** must be declared as **huge** because it is larger than 64K. Using the **huge** keyword instead of the standard huge memory model means that the price for using huge data is only paid for this one large item. Other data can be accessed quickly within the default segment. The pointer **pa** could be used to point to **a**. Any pointer arithmetic for **pa** (such as **pa++**) would be performed using 32-bit arithmetic.

```
char *pa;               /* small-model program */
char far *pb;
```

The pointer **pa** is declared as a near pointer to **char** in the example above. The pointer is near by default since the example appears in a small-model program. By contrast, **pb** is allocated as a far pointer to **char**; **pb** could be used to point to, and step through, an array of characters stored in a segment other than the default data segment. For example, **pa** might be used to point to array **a** in the first example, while **pb** might be used to point to array **b**.

```
char far * *pa;         /* small-model program */
char far * *pa;         /* large-model program */
```

The pointer declarations in the example above illustrate the interaction between the memory model chosen and the **near** and **far** keywords. Although the declarations for **pa** are identical, in a small-model program **pa** is declared as a near pointer to an array of far pointers to type **char**, while in a large-model program, **pa** is declared as a far pointer to an array of far pointers to type **char**.


```
char far * near *pb;          /* any model */  
char far * far *pb;
```

In the first declaration in the example above, `pb` is declared as a near pointer to an array of far pointers to type **char**; in the second declaration, `pb` is declared as a far pointer to an array of far pointers to type **char**. Note that, in this example, the **far** and **near** keywords override the model-specific addressing conventions shown in the example preceding the example above; the declarations for `pb` would have the same effect, regardless of the memory model.

6.4.3 Declaring Functions with the near and far Keywords

The rules for using the **near** and **far** keywords for functions are similar to those for using them with data, as specified below:

- The keyword always modifies the function or pointer immediately to its right. See Section 4.3.3, “Declarators with Special Keywords,” of the *Microsoft C Language Reference* for more information about rules for evaluating complex declarations.
- If the item immediately to the right of the keyword is a function, then the keyword determines whether the function will be allocated as near or far. For example,

```
char far fun( );
```

defines `fun` as a function called with a 32-bit address and returning type **char**.
- If the item immediately to the right of the keyword is a pointer to a function, then the keyword determines whether the function will be called using a near (16-bit) or far (32-bit) address. For example,

```
char (far * pfun)( );
```

defines `pfun` as a far pointer (32 bits) to a function returning type **char**.
- Function declarations must match function definitions.
- The **huge** keyword cannot be applied to functions.

■ Examples

```
void char far fun(void);           /* small model */
void char far fun(void)
{
    .
    .
    .
}
```

In the example above, `fun` is declared as a function returning type **char**. The **far** keyword in the declaration means that `fun` must be called with a 32-bit call.

```
static char far * near fun( );     /* large model */
static char far * near fun( )
{
    .
    .
    .
}
```

In the large-model example above, `fun` is declared as a near function that returns a far pointer to type **char**. Such a function might be seen in a large-model program as a helper routine that is used frequently, but only by the routines in its own module. Since all routines in a given module share the same code segment, the function could always be accessed with a near call. However, you could not pass a pointer to `fun` as an argument to another function outside the module in which `fun` was declared.

```
void far *fun(void);               /* small model */
void (far * pfun) ( ) = fun;
```

The small-model example above declares `pfun` as a far pointer to a function that has a **void** return type, and then assigns the address of `fun` to `pfun`. In fact, `pfun` could be used to point to any function accessed with a far call. Note that if the function pointed to by `pfun` has not been declared with the **far** keyword, or if it is not far by default, then calling that function through `pfun` would cause the program to fail.

```
double far * (far fun) ( );       /* compact model */
double far * (far *pfun) ( ) = fun;
```

The final example above declares `pfun` as a far pointer to a function that returns a far pointer to type **double**, and then assigns the address of `fun` to `pfun`. This might be used in a compact-model program for a function that is not used frequently and thus does not need to be in the default

code segment. Both the function and the pointer to the function must be declared with the **far** keyword.

6.4.4 Pointer Conversions

Passing pointers as arguments to functions may cause automatic conversions in the size of the pointer argument, since passing a pointer to a function forces the pointer size to the larger of the following two sizes:

- The default pointer size for that type, as defined by the memory model used during compilation.
For example, in medium-model programs, data pointer arguments are near by default, and code pointer arguments are far by default.
- The type of the argument.

If a function prototype with argument types is given, the compiler performs type checking and enforces the conversion of actual arguments to the declared type of the corresponding formal argument. However, if no declaration is present or the argument-type list is empty, the compiler will convert pointer arguments automatically to the larger of the default type or the type of the argument. To avoid mismatched arguments, you should always use a prototype with the argument types.

■ Examples

```
/* This program produces unexpected results in compact-,
** large-, or huge-model programs.
**/

main( )
{
    int near *x;
    char far *y;
    int z = 1;

    test_fun(x, y, z);    /* x will be coerced to far
                          ** pointer in compact, large,
                          ** or huge model
                          */
}

int test_fun(ptr1, ptr2, a)
    int near *ptr1;
    char far *ptr2;
    int a;
{
    printf("Value of a = %d\n", a);}
```

If the preceding example is compiled as a small-model program (no memory-model options or the `/AS` option on `CL` command line) or medium-model program (`/AM` option), the size of pointer argument `x` is 16 bits, the size of pointer argument `y` is 32 bits, and the value printed for `a` is 1. However, if the preceding example is compiled with the `/AC`, `/AL`, or `/AH` option, both `x` and `y` are automatically converted to far pointers when they are passed to `test_fun`. Since `ptr1`, the first parameter of `test_fun`, is defined as a near pointer argument, it takes only 16 bits of the 32 bits passed to it. The next parameter, `ptr2`, takes the remaining 16 bits passed to `ptr1`, plus 16 bits of the 32 bits passed to it. Finally, the third parameter, `a`, takes the left-over 16 bits from `ptr2`, instead of the value of `z` in the main function. This shifting process does not generate an error message, since both the function call and the function definition are legal, but in this case the program does not work as intended, since the value assigned to `a` is not the value intended.

To pass `ptr1` as a near pointer, you should include a forward declaration that specifically declares this argument for `test_fun` as a near pointer, as shown below:

```
/* First, declare test_fun so the compiler knows in advance
** about the near pointer argument:
*/
int test_fun(int near*, char far *, int);

main( )
{
    int near *x;
    char far *y;
    int z = 1;

    test_fun(x, y, z);    /* now, x will not be coerced
                          ** to a far pointer; it will be
                          ** passed as a near pointer,
                          ** no matter what memory
                          ** model is used
                          */
}

int test_fun(ptr1, ptr2, a)
    int near *ptr1;
    char far *ptr2;
    int a;
{
    printf("Value of a = %d\n", a);
}
```


Note that it would not be sufficient to reverse the definition order for `test_fun` and `main` in the first example to avoid pointer coercions; the pointer arguments must be declared in a forward declaration, as in the second example.

6.5 Creating Customized Memory Models

A third method of managing memory models is to combine features of the standard memory models to create your own customized memory model. You should have a thorough understanding of C memory models and the architecture of 8086 and 80286 processors before creating your own non-standard memory models, since there is no library support—other than the C start-up routines—for nonstandard memory models.

The `/Astring` option lets you change the attributes of the standard memory models to create your own memory models. The three letters in *string* correspond to the code pointer size, the data pointer size, and the stack- and data-segment setup, respectively. Because the letter allowed in each field is unique to that field, you can give the letters in any order after `/A`. All three letters must be present.

The standard-memory-model options (`/AS`, `/AM`, `/AC`, `/AL`, and `/AH`) can be specified in the `/Astring` form. As an example of how to construct memory models, the standard-memory-model options are listed below with their `/Astring` equivalents:

<u>Standard</u>	<u>Custom Equivalent</u>
<code>/AS</code>	<code>/Asnd</code>
<code>/AM</code>	<code>/Alnd</code>
<code>/AC</code>	<code>/Asfd</code>
<code>/AL</code>	<code>/Alfd</code>
<code>/AH</code>	<code>/Alhd</code>

As an example of the use of customized models, you might want to create a huge-compact model. This model would allow huge data items, but only one code segment. The option for specifying this model would be `/Ashd`.

An even more common use of customized models is to set up segments (see Section 6.5.3 for more information).

If you use a customized memory model for a program that includes both far and near functions, be aware of the following issues:

- The **chkstk** library function should be called only in functions that are compiled in the same model as the library being used. (For compatibility with XENIX, the **chkstk** function name cannot be model encoded.)
- The interfaces to floating-point function calls (generated when the **/FPc**, **/FPc87**, or **/FPa** option is used in compiling) are not model encoded, so the same restriction is placed on functions containing floating-point calls: they must be compiled with the same model as the library being used.

Note

For the purposes of the descriptions that follow, the letters **l** (for “long”) and **s** (for “short”) are used for code pointers to distinguish them in the memory-model string from the letters for data pointers.

6.5.1 Code Pointers

■ Options

/Asxx	Near code pointers
/Alxx	Far code pointers

The letter **s** tells the compiler to generate near (16-bit) pointers and addresses for all code items. This is the default for small- and compact-model programs.

The letter **l** means that far (32-bit) pointers and addresses are used to address all code items. Far pointers are the default for medium-, large-, and huge-model programs.

6.5.2 Data Pointers

■ Options

/Anxx	Near data pointers
/Afx	Far data pointers
/Ahxx	Huge data pointers

Three sizes are available for data pointers: near, far, and huge. The letter **n** tells the compiler to use near (16-bit) pointers and addresses for all data. This is the default for small- and medium-model programs.

The letter **f** specifies that all data pointers and addresses are far (32-bit). This is the default for compact- and large-model programs.

The letter **h** specifies that all data pointers and addresses are far (32-bit). This is the default for huge-model programs.

When far data pointers are used, no single data item may be larger than a segment (64K) because address arithmetic is performed only on 16 bits (the offset portion) of the address. When huge data pointers are used, individual data items can be larger than a segment (64K) because address arithmetic is performed on the entire 32 bits of the address.

6.5.3 Setting Up Segments

■ Options

/Adxx	Sets SS = DS
/Au[xx]	Sets SS != DS ; DS reloaded on function entry
/Aw[xx]	Sets SS != DS ; DS not reloaded on function entry

The letter **d** tells the compiler that the segment addresses stored in the **SS** and **DS** registers are equal; that is, the stack segment and the default data segment are combined into a single segment. This is the default for all programs. In small- and medium-model programs, the stack plus all data must occupy less than 64K; thus, any data item is accessed with only a 16-bit offset from the segment address in the **SS** and **DS** registers.

In compact-, large-, and huge-model programs, initialized global and static data are placed in the default data segment. The address of this segment is stored in the **DS** and **SS** registers. All pointers to data, including pointers to local data (the stack), are full 32-bit addresses. This is important to remember when passing pointers as arguments in large-model programs. Although you may have more than 64K of total data in these models, there can be no more than 64K of data in the default segment. The **/Gt** and **/ND** options can be used to control allocation of items in the default data segment if a program exceeds this limit. (See Section 6.6, "Setting the Data Threshold," and Section 6.7, "Naming Modules and Segments," for more information about these options.)

The letter **u** allocates different segments for the stack and the data segments. Each object file (module) is allocated its own segment for global and static data items. Note that the **/ND** option, described in Section 6.7, must be specified along with the letter **u** to allocate data segments other than the default. When the letter **u** is specified with **/ND**, the address in the **DS** register is saved upon entry to each function, and the new **DS** value for the module in which the function was defined is loaded into the register. The previous **DS** value is restored on exit from the function.

Therefore, only one data segment is accessible at any given time. The **/ND** option can be used to combine these segments into a single segment.

If a standard memory-model option precedes it on the command line, the **/Au** option can be specified without any letters indicating data- or code-pointer sizes. In this case, the program uses the specified memory model, but different segments are set up for the stack and data segments.

A single segment must be allocated for the stack, and its address stored in the **SS** register. The stack segment does not change throughout the entire program.

The letter **w**, like the letter **u**, sets up a separate stack segment, but does not automatically load the **DS** register at each module entry point. This option is typically used when writing application programs that interface with an operating system or with a program running at the operating-system level. The operating system or the program running under the operating system actually receives the data intended for the application program and places that data in a segment; then the operating system or program must load the **DS** register with the segment address for the application program.

As with the **/Au** option, the **/Aw** option can be specified without data- and code-pointer letters if a standard memory-model option precedes it on the command line. In this case, the program uses the specified memory model, but different segments are set up for the stack and data segments, and the **DS** register is not reloaded at each module entry point.

Even though **u** and **w** set up a separate segment for the stack, the stack's size is still fixed at the default size unless this is overridden with the **/F** compiler option or the **/STACK** linker option.

6.5.4 Library Support for Customized Memory Models

Most C programs make function calls to the routines in the C run-time library. Library support is provided for the five standard memory models (small, medium, compact, large, and huge) through four separate run-time libraries (huge and large models both use the large-model library). When you write mixed-model programs, you are responsible for determining which library (if any) is suitable for your program and for ensuring that the appropriate library is used.

Library support is provided for customized memory models where the stack and default data segments are combined into a single segment (**/Adxx**), but not for customized memory models where these segments are different (**/Auxx**, **/Awxx**, **/Au**, and **/Aw**). In the latter cases, you probably need to create a customized library to be used with your customized memory model. Use the **/NOD** (for “no default library search”) option when linking, and specify the library files and object files you want to use.

Be sure to use the start-up routine from the appropriate library for your memory model. Table 6.2 shows the libraries from which to extract the start-up routine for each customized memory model.

Table 6.2
Start-Up Routines for
Customized Memory Models

Memory-Model Option	Use Start-Up From Library
<code>/Asnx; /AS</code> plus <code>/Ax¹</code>	<code>SLIBCf.LIB²</code>
<code>/Asfx; /Ashx;</code> <code>/AC¹ plus /Ax</code>	<code>CLIBCf.LIB²</code>
<code>/Alnx; /AM</code> plus <code>/Ax¹</code>	<code>MLIBCf.LIB²</code>
<code>/Alfx; /Alhx;</code> <code>/AL plus /Ax₁</code> <code>/AH plus /Ax¹</code>	<code>LLIBCf.LIB²</code>

¹ Where *x* is either **u** or **w**

² Where *f* is either **E** (emulator library), **7** (8087/80287 library), or **A** (alternate math library)

In general, library functions do not support customized memory models, since a particular run-time routine may in turn call another library routine that conflicts with your customized model.

6.6 Setting the Data Threshold

■ Option

`/Gt[number]`

By default, the compiler allocates all static and global data items within the default data segment in the small and medium memory models. In compact-, large-, and huge-model programs, only *initialized* static and global data items are assigned to the default data segment. The `/Gt` option causes all data items whose size is greater than or equal to *number* bytes to be allocated to a new data segment. When *number* is specified, it must follow the `/Gt` option immediately, with no intervening spaces. When *number* is omitted, the default threshold value is 256. When the `/Gt` option is omitted, the default threshold value is 32,767.

You can use the `/Gt` option only with compact-, large-, and huge-model programs, since small- and medium-model programs have only one data segment. The option is particularly useful with programs that have more than 64K of initialized static and global data in small data items.

6.7 Naming Modules and Segments

■ Options

`/NM modulename`
`/NT textsegment`
`/ND datasegment`

“Module” is another name for an object file created by the C compiler. Every module has a name. The compiler uses this name in error messages if problems are encountered during processing. The module name is usually the same as the source-file name. You can change this name using the `/NM` (for “name module”) option. The new *modulename* can be any combination of letters and digits. The space between `/NM` and *modulename* is optional.

A “segment” is a contiguous block of binary information (code or data) produced by the C compiler. Every module has at least two segments: a text segment containing the program instructions, and a data segment containing the program data. Each segment in every module has a name. The linker uses this name to define the order in which the segments of the program appear in memory when loaded for execution. (Note that the segments in the group named **DGROUP** are an exception; see the *Microsoft Mixed-Language Programming Guide* for more information.)

Text and data segment names normally are created by the C compiler. These default names depend on the memory model chosen for the program. For example, in small-model programs the text segment is named **_TEXT** and the data segment is named **_DATA**. These names are the same for all small-model modules, so all text segments from all modules are loaded as one contiguous block, and all data segments from all modules form another contiguous block.

In medium-model programs, the text from each module is placed in a separate segment with a distinct name, formed by using the module base name along with the suffix **_TEXT**. The data segment is named **_DATA**, as in the small model.

In compact-model programs, the data from each module are placed in a separate segment with a distinct name, formed by using the module base name along with the suffix **_DATA**. The exception to this is initialized

global and static data, which are put in the default data segment, **_DATA**. The code segment is named **_TEXT**, as in the small model.

In large- and huge-model programs, the text and data from each module are loaded into separate segments with distinct names. Each text segment is given the name of the module plus the suffix **_TEXT**. The data from each segment are placed in a private segment with a unique name (except for initialized global and static data placed in the default data segment). The naming conventions for text and data segments are summarized in Table 6.3.

Table 6.3
Segment-Naming Conventions

Model	Text	Data	Module
Small	_TEXT	_DATA	<i>filename</i>
Medium	<i>module</i> _TEXT	_DATA	<i>filename</i>
Compact	_TEXT	_DATA ¹	<i>filename</i>
Large	<i>module</i> _TEXT	_DATA ¹	<i>filename</i>
Huge	<i>module</i> _TEXT	_DATA ¹	<i>filename</i>

¹ Name of default data segment; other data segments have unique private names.

You can override the default names used by the C compiler (thus overriding the default loading order) by using the **/NT** (for “name text”) and **/ND** (for “name data”) options. These options set to a given name the names of the text and data segments in each module being compiled. The *textsegment* argument used with the **/NT** option and the *datasegment* argument used with the **/ND** option can be any combination of letters and digits. The space between **/NT** and *textsegment*, and the space between **/ND** and *datasegment*, are optional.

If you use the **/ND** option to change the name of the default data segment, your program can no longer assume that the address contained in the stack segment register (**SS**) is the same as the address in the data segment register (**DS**). You must therefore compile your program either with the **/Astring** form of the memory-model option and the **u** option for the segment-setup letter, or with the **/A** option for a standard memory model followed by the **/Au** option as in the following example:

```
CL /As /Au /ND DATA1 PROG1.C
```

Use of the **/Au** option forces the compiler to generate code to load **DS** with the correct data-segment value on entry to the code. See Section 6.5,

“Creating Customized Memory Models,” for more information on the */Astring* options. All modules whose data segments have the same name have these segments combined into a single segment named DATA1 at link time.

6.8 Specifying Text and Data Segments

■ Pragmas

```
# pragma alloc_text (textsegment, function1[, function2]...)
# pragma same_seg (variable1[, variable2]...)
```

The **alloc_text** pragma gives you source-level control over the segment in which particular functions are allocated. The **same_seg** pragma provides information the compiler can use to generate better code.

If you use overlays or swapping techniques to handle large programs, **alloc_text** allows you to tune the contents of their text segments for maximum efficiency. The **alloc_text** pragma must appear before the definitions of any of the specified functions, but it may appear either before or after the functions are declared or called. Any functions specified in an **alloc_text** pragma must either be explicitly declared with the **far** keyword or assumed to be far because of the memory model used (medium, large, or huge).

The **same_seg** pragma tells the compiler to assume that the specified external variables are allocated in the same data segment. You are responsible for making sure that these variables are put in the same data segment; one way to do this is to specify the **/ND** option when you compile the program. The **same_seg** pragma must appear before any of the specified variables is used in executable code and after the variables are declared. Variables specified in a **same_seg** pragma must be explicitly declared with **extern** storage class, and they must either be explicitly declared with the **far** keyword or assumed to be far because of the memory model used (compact, large, or huge).

CHAPTER

CONTROLLING FLOATING- POINT MATH OPERATIONS

7

7.1	Introduction.....	163
7.2	Summary of Math Packages	163
7.2.1	The Emulator Package.....	163
7.2.2	The 8087/80287 Package.....	164
7.2.3	The Alternate Math Package	164
7.3	Selecting Floating-Point (/FP) Options	165
7.3.1	The /FPi Option	167
7.3.2	The /FPi87 Option	168
7.3.3	The /FPc Option.....	168
7.3.4	The /FPc87 Option	169
7.3.5	The /FPa Option.....	169
7.4	Library Considerations for Floating-Point Options.....	170
7.4.1	In-Line Instructions or Calls	170
7.4.2	Using One Standard Library for Linking.....	170
7.5	Compatibility between Floating-Point Options	173
7.6	Using the NO87 Environment Variable.....	174
7.7	If Your Computer Is Not IBM Compatible.....	175

7.1 Introduction

This chapter discusses the various ways that you can control how your Microsoft C programs handle floating-point math operations. It describes the math packages that you can include in C libraries when you run the **SETUP** program, then discusses the **CL** command options for choosing the appropriate library for linking and controlling floating-point instructions.

This chapter also explains how to override floating-point options by changing libraries at link time, and how to control use of an 8087 or 80287 coprocessor through the **NO87** environment variable.

7.2 Summary of Math Packages

The Microsoft C Compiler offers a choice of the following three math packages for handling floating-point operations:

1. Emulator (default)
2. 8087/80287
3. Alternate math

When you run the **SETUP** program, you choose one of these three math packages. **SETUP** includes the math package you choose in the library it builds. Any programs that are linked with that library use the math package included in the library; you must use the appropriate **CL** option to make sure that the library you want is used at link time.

The following descriptions of these math packages are designed to help you choose the appropriate math option for your needs when you build a library using **SETUP**.

7.2.1 The Emulator Package

The emulator package uses an 8087 or 80287 coprocessor if one is installed. If no coprocessor is installed, the emulator provides many 8087/80287 functions in software. This is the default math package; **SETUP** uses it if you do not explicitly choose another package.

The emulator package is the best choice if you want to maximize accuracy in program results and if the program will be run on systems with and without coprocessors.

The emulator package can perform basic operations to the same degree of accuracy as an 8087/80287. However, the emulator routines used for transcendental math functions differ slightly from the corresponding 8087/80287 functions, and this difference can cause a slight difference (usually within two bits) in the results of these operations when performed with the emulator instead of with an 8087/80287.

Important

When you use an 8087 or 80287 coprocessor or the emulator, interrupt-enable, precision, underflow, and denormalized-operand exceptions are masked by default. The remaining exceptions are unmasked. See Section E.4.2, "Other Run-Time Error Messages," and the discussion of the `_control87` function in the *Microsoft C Run-Time Library Reference* for more information about 8087 floating-point exceptions.

7.2.2 The 8087/80287 Package

The 8087/80287 math package allows you to use an 8087 or 80287 coprocessor to perform floating-point operations. You must have an 8087 or 80287 installed to use this package. This package gives you the fastest, smallest programs possible for handling floating-point math.

7.2.3 The Alternate Math Package

The alternate math package gives you the smallest and fastest programs you can get without a coprocessor. However, the program results are not as accurate as results given by the emulator package.

The alternate math package uses a subset of the Institute of Electrical and Electronics Engineers, Inc. (IEEE) standard-format numbers; infinities, NaNs, and denormal numbers are not used.

7.3 Selecting Floating-Point (/FP) Options

■ Options

/FPa	Generates floating-point calls; selects <i>mLIBCA.LIB</i>
/FPc	Generates floating-point calls; selects <i>mLIBCE.LIB</i>
/FPc87	Generates floating-point calls; selects <i>mLIBC7.LIB</i>
/FPi	Generates in-line instructions; selects <i>mLIBCE.LIB</i> (default)
/FPi87	Generates in-line instructions; selects <i>mLIBC7.LIB</i>

The **/FP** options of the **CL** command control how a program will handle floating-point math. You can use only one of these options on the **CL** command line. The option applies to the entire command line, regardless of the option's position.

Each **/FP** option includes two parts, which specify the following:

1. How floating-point instructions are included in the program: by using in-line 8087/80287 instructions or calls to floating-point library functions. The letter **i** indicates in-line instructions; the letters **c** and **a** indicate floating-point calls.
2. Which floating-point package is selected by default when you link.

Based on the **/FP** option and the memory-model option you choose, the **CL** command embeds a library name in the object file that it creates. (See Table 3.1 in Section 3.3.1, "Memory-Model and Floating-Point Options," for a list of the library names used for each combination.) This library is then considered the default library; that is, the linker searches in the standard places for a library with that name. If it finds a library with that name, the linker uses the library to resolve external references in the object file being linked. Otherwise, it displays a message indicating that it could not find the library.

This mechanism allows the linker to link object files with the appropriate library automatically. However, as explained later in this section and in Section 7.4, "Library Considerations for Floating-Point Options," you are allowed to link with a different library in some cases.

Table 7.1 summarizes the **/FP** options and their effects.

Table 7.1
Summary of Floating-Point Options

Option	Method	Advantages	Use of Coprocessor	Combined Libraries Selected
/FPI	In-line	Default; larger than /FPI87 , but can work without coprocessor; most efficient way to get maximum precision without a coprocessor	Uses coprocessor if present ¹	mLIBCE.LIB ²
/FPI87	In-line	Smallest and fastest option available with a coprocessor	Requires coprocessor unless library changed at link time ¹	mLIBC7.LIB ³
/FPc	Calls	Slower than /FPI , but allows use of alternate math library at link time	Uses coprocessor if present ¹	mLIBCE.LIB ^{2,4}
/FPc87	Calls	Slower than /FPI87 , but allows use of alternate math library at link time	Requires coprocessor unless library changed at link time ¹	mLIBC7.LIB ^{3,4}
/FPa	Calls	Fastest and smallest option available without coprocessor, but sacrifices some accuracy for speed	Ignores coprocessor	mLIBCA.LIB ^{2,3}

¹ Use of the coprocessor can be suppressed by setting **NO87**.

² Can be linked explicitly with **mLIBC7.LIB** at link time

³ Can be linked explicitly with **mLIBCE.LIB** at link time

⁴ Can be linked explicitly with **mLIBCA.LIB** at link time

The remainder of this section discusses the **/FP** options and the advantages and disadvantages of each option.

Note

Some expressions may be evaluated at compile time. Such evaluations always use the highest precision possible and are unaffected by the floating-point option you choose. The **/AS** (small) memory-model option is the default. Therefore, if no memory-model option is given on the same **CL** command line, the default library for each floating-point option is **SLIBC*f*.LIB** (where *f* is **7**, **E**, or **A**, depending on the math package the library supports).

7.3.1 The **/FPi** Option

The **/FPi** option generates in-line instructions for an 8087 or 80287 coprocessor and places the name of the emulator library (**mLIBCE.LIB**) in the object file. At link time, you can specify the 8087/80287 library (**mLIBC7.LIB**) instead. If you do not choose a floating-point option, **CL** uses the **/FPi** option by default.

The **/FPi** option is particularly useful if you do not know whether an 8087 or 80287 coprocessor will be available at run time. Programs compiled with **/FPi** work as described below:

- If a coprocessor is present at run time, the program uses the coprocessor.
- If no coprocessor is present, the program uses the emulator. In this case, the **/FPi** option offers the most efficient way to get maximum precision in floating-point results.

The Microsoft C Optimizing Compiler does not generate “true” in-line 8087/80287 instructions: instead, it generates software interrupts to library code, which then fixes up the interrupts to use either the emulator or the coprocessor, depending on whether or not a coprocessor is present. The fix-ups can be removed by simply assembling the following program and linking it with the C program:


```

public FIARQQ, FICRQQ, FIDRQQ, FIERQQ, FISRQQ, FIWRQQ
public FJARQQ, FJCRQQ, FJSRQQ

FIARQQ      EQU    0
FICRQQ      EQU    0
FIDRQQ      EQU    0
FIERQQ      EQU    0
FISRQQ      EQU    0
FIWRQQ      EQU    0
FJARQQ      EQU    0
FJCRQQ      EQU    0
FJSRQQ      EQU    0

END

```

Assembling and linking this program with C programs can save execution time (the time required to fix up all the interrupts the first time). However, a C program linked with this program will run only if a coprocessor is present. (This option is useful if you are developing programs to be run from read-only memory; see Appendix D, "Writing Programs for Read-Only Memory," for more information.)

7.3.2 The /FPi87 Option

The **/FPi87** option includes the name of an 8087/80287 library (**mLIBC7.LIB**) in the object file. At link time, you can specify an emulator library (**mLIBCE.LIB**) instead.

If you use the **/FPi87** option and link with **mLIBC7.LIB**, an 8087 or 80287 coprocessor *must* be present at run time; otherwise, the program fails and the following error message is displayed:

```

run-time error R6002
- floating point not loaded

```

If you compile with **/FPi87** and link with **mLIBCE.LIB**, you can set the **NO87** environment variable to suppress the use of the coprocessor. (See Section 7.6 for a description of **NO87**.)

Compiling with the **/FPi87** option results in the smallest, fastest programs possible for handling floating-point results.

7.3.3 The /FPc Option

The **/FPc** option generates floating-point calls to the emulator library and places the names of an emulator library (**mLIBCE.LIB**) in the object file. At link time, you can specify an 8087/80287 library (**mLIBC7.LIB**) or alternate math library (**mLIBCA.LIB**) instead. Thus, the **/FPc** option gives you more flexibility than the **/FPi** option in the libraries you can use for linking.

The **/FPc** option is also recommended in the following cases:

- If you compile modules that perform floating-point operations and plan to include these modules in a library
- If you compile modules that you want to link with libraries other than the libraries provided with the Microsoft C Optimizing Compiler

7.3.4 The **/FPc87** Option

The **/FPc87** option generates function calls to routines in the 8087/80287 library (**mLIBC7.LIB**) that perform the corresponding 8087/80287 instructions. As with the **/FPi87** option, you can change your mind at link time and link with an emulator library (**mLIBCE.LIB**); however, you have more flexibility in choosing libraries, since you can change your mind and link with the appropriate alternate math library as well (**mLIBCA.LIB**).

You must have an 8087 or 80287 coprocessor installed in order to run programs compiled with the **/FPc87** option and linked with an 8087/80287 library. Otherwise, the program fails and the following error message is displayed:

```
run-time error R6002
- floating point not loaded
```

Note

Certain optimizations are not performed when **/FPc87** is used. This may reduce the efficiency of your code; and, since arithmetic of different precision may result, there may be slight differences in your results.

7.3.5 The **/FPa** Option

The **/FPa** option generates floating-point calls and selects the alternate math library for the appropriate memory model (**mLIBCA.LIB**). Calls to this library provide your fastest and smallest option if you do not have an 8087 or 80287 coprocessor. With this option, you can change your mind at link time and use an emulator library (**mLIBCE.LIB**) or 8087/80287 library (**mLIBC7.LIB**).

7.4 Library Considerations for Floating-Point Options

You may want to use libraries in addition to the default library for the floating-point option you have chosen on the **CL** command line. For example, you may want to create your own libraries (or other collections of subprograms in object-file form), then link these libraries at a later time with object files that you have compiled using different **CL** options.

The following paragraphs discuss these cases and how to handle them. Although the discussion assumes that you are putting your precompiled object files into libraries, the same considerations apply if you are simply using individual object files.

7.4.1 In-Line Instructions or Calls

First, you should decide whether you want to use in-line instructions and compile with the **/FPi87** or **/FPi** option, or floating-point function calls and compile with the **/FPc87**, **/FPc**, or **/FPa** option.

If you choose in-line instructions for your precompiled object files, you cannot link with an alternate math library (**mLIBCA.LIB**). However, in-line instructions give the best performance from your programs on machines that have an 8087 or 80287 coprocessor installed.

If you choose calls, your programs are slower, but at link time you can use any standard C library—that is, any library created by the **SETUP** program—that supports the memory model you have chosen.

7.4.2 Using One Standard Library for Linking

You must also be sure that you use only one standard C library when you link. You can control which library is used in one of two ways:

1. At link time, as the *first* name in the list of object files to be linked, give an object file that has the name of the desired library. For example, if you want to use an alternate math library, give the name of an object file compiled using the **/FPa** option. All floating-point calls in this object file refer to the alternate math library.
2. At link time, give the **/NOD** (no default library search) option and then specify the name of the combined library file you want to use in the *link-libinfo* field of the **CL** command line. This overrides the library names embedded in the object files, and all floating-point calls refer to the libraries you specify.

Deciding how to link with the correct libraries can become complicated since each library name mentioned in one of the object files being linked is added to the “linker search list” (the list of libraries that the linker searches).

For example, suppose the following:

- You have used the **/FPa** option to compile a set of object files.
- Each of these object files includes a default library name (that is, you did not use the **/Zl** option to compile).
- You have used the **LIB** utility (described in Chapter 13 of the Microsoft CodeView and Utilities manual) to combine these object files into a library.
- You want to link the library you have created with an object file that was created using the **/FPc87** option.

At link time, the **SLIBC7.LIB** and **SLIBCA.LIB** libraries are both in the linker search list (assuming you compiled with the default memory-model option): **SLIBC7.LIB** because this name is embedded in the object file you are linking, and **SLIBCA.LIB** because this name is embedded in the object files that constitute the library. The linker first searches the libraries named in the object file you are linking, so it searches **SLIBC7.LIB** before it searches **SLIBCA.LIB**. Since **SLIBC7.LIB** would resolve all external references correctly, this mechanism works correctly.

To ensure that they are used, the names of libraries that you want to link with can be specified in the *link-libinfo* field of the **CL** command line (as noted in method 2 above). In this case, the linker always searches the library you give on the command line before it searches any libraries named in the object files. However, you must make sure that you specify this library *after* any of your own libraries on the command line. If you don't, and your library contains a different search directive, you may encounter problems.

As an example of the problems you may encounter, assume the following scenario:

- The object modules in your library named B were compiled with the **/FPc87** option, so that each module contains search directives for **SLIBC7.LIB**.
- You are linking an object file named A that was compiled with the **/FPa** option, so that this object file contains a search directive for **SLIBCA.LIB**.
- You used the following command line to link your library B with the object file A:

```
CL A /link SLIBC7.LIB B
```


In this example, the linker searches libraries in the following order:

1. **SLIBC7.LIB** (since it is specified first on the command line)
2. **B** (since it is specified second on the command line)
3. **SLIBCA.LIB** (since **A**, the object module that you are linking, contains a search directive for this library)
4. **SLIBC7.LIB** (since the modules in **B**, your library, contain search directives for this library)

The linker would search for floating-point libraries as follows:

1. The linker searches **SLIBC7.LIB** and resolves references in the object file **A** to floating-point math routines and standard-library routines.
2. The linker closes **SLIBC7.LIB** and searches the next library in the list to satisfy references to routines in your library **B**. These routines normally contain references to standard run-time routines. Since **SLIBCA.LIB** is the next library to be searched, this library satisfies the references in **B**. However, this is not the library you intended to use, since you compiled **B** with the **/FPc87** option, which uses **SLIBC7.LIB** to resolve references to standard run-time routines.

As indicated in this example, you cannot mix libraries in this way, and you may get linker errors if you try. Note that if you had specified **B SLIBC7.LIB** instead of **SLIBC7.LIB B** on the **CL** command line, the linker would have searched **SLIBC7.LIB** instead of **SLIBCA.LIB** to resolve floating-point references in **B**, and the linking operation would have proceeded correctly.

To avoid this kind of ambiguity and make absolutely sure that you are specifying the correct standard library for linking, use the **/NOD** linker option. This option causes the linker to search only the libraries you specify on the command line.

Perhaps the safest course of all, especially when you are distributing libraries to others, is to compile the object files that make up the library with the **/Zl** option. This option tells the compiler not to include search directives in the object files. Later on, when you link the library with different object files, the standard library used for linking depends only on the floating-point and memory-model options used to compile the later object files. The **/FPc** compiler option is recommended for maximum flexibility in linking with such libraries.

■ Examples

```
CL CALC.C ANOTHER SUM
```

In the example above, the source file `CALC.C` is compiled with the default floating-point option, `/FPi`. The `/FPi` option generates in-line instructions and selects the small-model emulator combined library (`SLIBCE.LIB`) since no floating-point option is given and the small-model library is the default.

```
CL /FPa CALC.C ANOTHER SUM /link SLIBCE.LIB /NOD
```

In the example above, `CALC.C` is compiled with the alternate math option (`/FPa`). The `/link` option specifies the `/NOD` option so that the `SLIBCA.LIB` library (whose name is embedded in the object file `CALC.OBJ`) is not searched. This option specifies the name `SLIBCE.LIB` instead so that all floating-point calls refer to the standard small-model emulator library instead of the alternate math library.

```
CL /FPc87 CALC.C ANOTHER.OBJ SUM.OBJ /link SLIBCA.LIB /NOD
```

In the example above, `CALC.C` is compiled with the `/FPc87` option, which selects the `SLIBC7.LIB` library. The `/link` option overrides the default library specification, since the `/NOD` option and the name of the alternate math library (`SLIBCA.LIB`) are specified.

7.5 Compatibility between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link two or more source files to produce an executable program file, you are responsible for ensuring that floating-point operations are handled in a consistent way and that the environment is set up properly to allow the linker to find the required library. See Section 2.4.5 for information about setting up your environment, Section 3.3.1 for information about choosing floating-point options for the libraries you build with the **SETUP** program, and Chapter 12 of the Microsoft CodeView and Utilities manual for a detailed discussion of linking.

Note

If you are building libraries of C routines that contain floating-point operations, the **/FPc** floating-point option is recommended for all compilations. The **/FPc** option offers the greatest flexibility.

■ Examples

```
CL /AM CALC.C ANOTHER SUM /link MLIBC7 /NOD
```

The example above compiles the program **CALC.C** with the medium-model option (**/AM**). Because no floating-point option is specified, the default, **/FPi**, is used. The **/FPi** option generates 8087/80287 instructions and specifies the emulator library **MLIBCE.LIB** in the object file. The **/link** field specifies the **/NOD** option and the names of the medium-model 8087/80287 library. Specifying the 8087/80287 library forces the program to use an 8087 coprocessor; the program fails if a coprocessor is not present.

```
CL /FPa CALC.C ANOTHER SUM /link SLIBCE /NOD
```

The example above compiles **CALC.C** using the small (default) memory model and the alternate math option (**/FPa**). The **/link** field specifies the **/NOD** option and the library name **SLIBCE.LIB**. Specifying the emulator library causes all floating-point calls to refer to the emulator library instead of the alternate math library.

```
CL /FPc87 CALC.C ANOTHER SUM /link SLIBCA.LIB/NOD
```

The example above compiles **CALC.C** with the **/FPc87** option, which places the library name **SLIBC7.LIB** in the object file. The **/link** field overrides this default-library specification by giving the **/NOD** option and the names of the small-model alternate math library (**SLIBCA.LIB**).

7.6 Using the NO87 Environment Variable

Programs compiled using the **/FPc** or **/FPi** option automatically use an 8087 or 80287 coprocessor at run time if one is installed. You can override this and force the use of the emulator instead by setting an environment variable named **NO87**.

If **NO87** is set to any value when the program is executed, use of the coprocessor is suppressed. The value of the **NO87** setting is printed on the standard output as a message. The message is printed only if a coprocessor is present and suppressed; if no coprocessor is present, no message appears. If you don't want a message to be printed, set **NO87** equal to one or more spaces.

Note that only the presence or absence of the **NO87** definition is important in suppressing use of the coprocessor. The actual value of the **NO87** setting is used only for printing the message.

The **NO87** variable takes effect with any program linked with an emulator library (*mLIBCE.LIB*). It has no effect on programs linked with 8087/80287 libraries (*mLIBC7.LIB*) or programs linked with alternate math libraries (*mLIBCA.LIB*).

■ Examples

SET NO87=Use of coprocessor suppressed

The example above causes the message *Use of coprocessor suppressed* to appear when a program is executed that uses an 8087 or 80287 coprocessor while an 8087 or 80287 coprocessor is present.

SET NO87=space

The example above sets the **NO87** variable to the space character. Use of the coprocessor is still suppressed, but no message is displayed.

7.7 If Your Computer Is Not IBM Compatible

The exception handler in the libraries for 8087 or 80287 floating-point calculations (*mLIBCE.LIB* and *mLIBC7.LIB*) is designed to work without modification on the IBM PC family of computers, and on closely compatible computers, including the Wang® PC, the AT&T® 6300, and the Olivetti® personal computers. Also, the libraries need not be modified for the Texas Instruments® Professional Computer, even though it is not compatible. Any machine that uses nonmaskable interrupts (NMI) for 8087/80287 exceptions should work with the unmodified libraries. However, if your computer is not one of these, and if you are not sure whether it is completely compatible, you may need to modify the 8087/80287 libraries.

All Microsoft languages that support the 8087 and 80287 coprocessors intercept 8087/80287 exceptions in order to produce accurate results and properly detect error conditions.

To make the libraries work correctly on noncompatible machines, you can modify the libraries. To make this easier, an assembly-language source file, **EMOEM.ASM**, is included on the distribution disk. Any machine that sends the 8087/80287 exception to an 8259 Priority Interrupt Controller (master or master/slave) should be easily supported by a simple table change to the **EMOEM.ASM** module. The source file contains further instructions on how to modify **EMOEM.ASM** and patch libraries and executable files.

CHAPTER

8

IMPROVING PROGRAM SPEED

8.1	Introduction.....	179
8.2	Using Register Variables	179
8.3	Optimization Options and Pragmas.....	181
8.3.1	Default Optimization	181
8.3.2	Generating Intrinsic Functions.....	181
8.3.3	Relaxing Alias Checking.....	182
8.3.4	Performing Loop Optimizations	182
8.3.5	Removing Stack Probes.....	183
8.3.6	Maximum Optimization	183
8.4	Choosing the Function-Calling Convention.....	183
8.5	Efficiency in Large Data Models.....	184
8.5.1	Changing Addressing with near, far, and huge Keywords	184
8.5.2	Setting the Data Threshold	185
8.5.3	Controlling Segments Used for Allocation	185
8.6	Efficiency in Large Code Models	185

8.1 Introduction

This chapter describes a number of ways that you can improve the execution speed of programs compiled with the Microsoft C Optimizing Compiler. These techniques include the following:

- Using register variables
- Using optimization options and pragmas
- Choosing function-calling conventions
- Choosing and adjusting memory models

Where applicable, this chapter discusses the interactions between these techniques and the trade-offs involved in using them.

8.2 Using Register Variables

One common way to write a program for maximum speed is to declare selected local (**auto**) variables with **register** storage class. The declaration of a register variable requests the compiler to use machine registers when allocating space for the variable, if possible. The **register** storage class can be specified for any variable, but **register** specifications are ignored except for variables of type **int** or **short** or for pointer types that are the same size as type **int**.

Up to two register variables may be allocated per function. In lexical order, the compiler takes the first two variables with **register** storage class that meet the size criteria. Any later requests for **register** storage class are ignored, so be sure to declare the most important register variables first. You may also want to declare register variables in parallel scope to achieve the effect of having more than two register variables per function.

The Microsoft C Optimizing Compiler automatically uses registers for variables within loops. Using register declarations for such variables may interfere with optimal loop code; you can experiment with various combinations of register and nonregister declarations to determine which combinations give the best results.

Register declarations can be used effectively for values, especially pointers, that appear outside of loops. Since a certain amount of code is required to save and restore registers, register declarations must be applied to values that are accessed at least three times within a function to cause any improvement in program speed.

■ Example

```

find_string(arr_of_chars, string)
char *string;
char *arr_of_chars[];
{
    int ix = 0;
    register char *q;
    while (*(q = string)) { /* string is not null */
        {
            register int i = ix;

            /* search for entry whose first character
             * matches first character of string, if any
             */

            while (i < MAX_ARR_SIZE && *arr_of_chars[i] != *q)
                i++;
            if (i == MAX_ARR_SIZE)
                return(1); /* no matching entry */
            ix = i;
        }

        /* we've found an entry in arr_of_chars which
         * might match string */

        {
            register char *p = arr_of_chars[ix];
            while (*p && *q && *p++ == *q++)
                ;
            if ((*p - *q) == 0)
                return(0) /* they match, return 0 */
            /* otherwise continue checking for possible
             * matches
             */
        }
    }
}

```

In the example above, the function named `find_string` actually has three register variables: `q`, `i`, and `p`. The function can use all three variables because `i` is through being used by the time `p` is needed. Simply introducing the `ix` variable to save the pointer from block to block speeds execution considerably because most work is being done in register variables.

8.3 Optimization Options and Pragmas

The **CL** compiler/linker driver provides a number of optimization options (**/O**, followed by one or more letters) that can improve program speed. In addition, the Microsoft C Optimizing Compiler includes several pragmas that allow you to control some of these optimizations on a local basis within a source program. The following sections outline these **CL** options and pragmas and their effects.

8.3.1 Default Optimization

If no **/O** option is given, the compiler uses the **/Ot** option, which optimizes programs for execution speed. However, this option does not enable loop optimizations or intrinsics. Some optimizations, such as long shifts, may be performed in line rather than using helper functions.

8.3.2 Generating Intrinsic Functions

The **/Oi** option generates intrinsic forms of the following functions:

- **memset, memcpy, memcmp**
- **strset, strcpy, strcmp, strcat**
- **inp, outp**
- **_rotr, _rotr, _lrotr, _lrotr,**
- **min, max, abs**

Intrinsics may be generated as in-line code or with different calling sequences. In general, using intrinsics increases program size but improves program speed. Note that the intrinsic forms of some functions may have slightly different semantics: for example, the intrinsic form of the **memcpy** function in compact- and large-model programs cannot handle huge arrays, but the function form can.

As with **/Ot**, this option may increase program size due to the additional code generated in line for each function. However, program execution is faster because no instructions for calling and returning from functions need to be performed.

The **intrinsic** pragma can be used to specify intrinsic functions on a local basis for any of the functions listed above. See Section 3.3.13.1 under the heading “Generating Intrinsic Functions” for information about the use of this pragma.

8.3.3 Relaxing Alias Checking

The **a** option letter can be used with the **l**, **s**, or **t** option letter to relax the assumptions the compiler makes about the use of “aliases” in the program. Use of the **/Oa** option can reduce the size of executable files and speed program execution. Its use is especially recommended when you also specify the **/Ol** option, since the compiler can detect a number of loop optimizations when the **/Oa** option is in effect that it cannot detect when **/Oa** is not in effect. However, before you specify **/Oa**, you must make sure that your program does not use multiple aliases to refer to the same memory location either directly or indirectly. For example, a program might do this indirectly in functions that operate on a communal variable and a pointer argument, or on multiple pointer arguments.

The **/Oa** option can be specified safely for programs that include calls to functions with address-type arguments. In this case, the compiler assumes that all variables whose addresses are passed to the function are modified, even if **/Oa** is specified.

In the cases noted above, the use of **/Oa** is most likely to cause incorrect optimizations within basic blocks (where most optimizations are applied) and within whole loop bodies (where loop optimizations are applied). In these cases, **/Oa** can still be specified safely even if aliases are used in the program, provided that no memory location is referenced by more than one name within any basic block or (if loop optimization is enabled) any loop body.

For more information and specific examples, see Section 3.3.13.1 under the heading “Relaxing Alias Checking.”

8.3.4 Performing Loop Optimizations

The **/Ol** option tells the compiler to perform loop optimizations. For best performance, use **/Ol** in conjunction with the **a** option letter (**/Oal**), which relaxes the assumptions the compiler makes about the use of aliases in the program. Using **/Oal** instead of just **/Ol** allows the compiler to detect many loop optimizations that it could not otherwise detect. (See Section 3.3.13.1 for information about possible restrictions on the uses of the **/Oa** option.)

You can control loop optimization on a local basis by specifying the **loop_opt** pragma. Loop optimization is turned off for any functions following **#pragma loop_opt(off)** and turned on for any functions following **#pragma loop_opt(on)** in a source program. This pragma overrides any loop optimization specified on the **CL** command line.

8.3.5 Removing Stack Probes

The **/Gs** option, described in Section 3.3.13.2, speeds program execution slightly by removing calls to stack-checking routines known as “stack probes.” Stack probes verify that a program has enough stack space to allocate required local variables. The potential disadvantage in removing stack probes is that stack-overflow errors may occur without generating a diagnostic message. However, this technique can be useful for programs that are known not to exceed the available stack space.

You can also control stack checking on a local basis by specifying the **check_stack** pragma. Stack checking is turned off for any functions following a **#pragma check_stack(off)** and turned on for any functions following a **#pragma check_stack(on)** pragma in the source program. This pragma overrides the stack checking (or removal of stack checking) specified on the **CL** command line.

8.3.6 Maximum Optimization

The **/Ox** option combines all of the optimization options described in Sections 8.3.1 through 8.3.4. Provided that the restrictions outlined for each optimization option do not apply, you can use the **/Ox** option to create the fastest possible program.

8.4 Choosing the Function-Calling Convention

Because C functions can accept a variable number of arguments, arguments passed to these functions must be pushed on the stack from right to left, with the first argument in the list being the last one pushed. In addition, the calling function, rather than the called function, is responsible for removing arguments from the stack.

This convention results in somewhat slower programs than the alternative convention used by Microsoft FORTRAN and Microsoft Pascal. In the FORTRAN/Pascal convention, arguments are pushed on the stack from left to right, in the order in which they are passed to the function, and the called function removes arguments from the stack. Since the code for removing arguments appears only once (in the called function) for the FORTRAN/Pascal convention, rather than multiple times (every time a function is called) as in the C convention, and since most programs have fewer functions than function calls in a program, the FORTRAN/Pascal calling convention usually results in smaller, faster programs.

You can specify the FORTRAN/Pascal calling convention for all functions in a module by compiling with the **/Gc** option. The trade-off for improved program speed is that you cannot call functions that use the C calling convention or take variable numbers of arguments unless you declare these functions, or pointers to these functions, with the **cdecl** keyword, which specifies the normal C calling conventions for these functions.

If you do not want to specify the FORTRAN/Pascal convention for a whole module, you can declare individual functions or pointers to functions with the **pascal** or **fortran** keyword. Either of these keywords tells the compiler that the function uses the FORTRAN/Pascal calling conventions.

8.5 Efficiency in Large Data Models

Programs are most efficient when their data reside in the default data segment: that is, when the data can be accessed with 16-bit (near) addresses. The Microsoft C Optimizing Compiler provides two standard memory models in which all data reside in the default data segment: the small (default) model and the medium model. The customized memory models that use near data pointers (**/Anxx**) also restrict program data to the default data segment. Programs compiled with these models are restricted to 64K of total data.

For programs compiled with the compact, large, and huge memory models, the compiler creates a default data segment containing all initialized global and static data and creates an additional data segment for each program module. Since accessing data outside the default data segment is slower than accessing data within the default data segment, programs will run faster if as many of their variables as possible are declared in such a way that they are allocated in the default data segment. One way to accomplish this is to initialize variables at the time you declare them. Sections 8.5.1 through 8.5.3 discuss other ways of controlling the allocation of data for large data models.

8.5.1 Changing Addressing with **near**, **far**, and **huge** Keywords

The **near**, **far**, and **huge** keywords allow you to explicitly specify the addressing used for particular data items and functions. These keywords override the default addressing conventions specified by the program's memory model. Thus, you can use them to improve the speed of access to program data. For example, you can tell the compiler to allocate data items in the default data segment for a compact-, large-, or huge-model program by declaring the items (or pointers to the items) with the **near** keyword. Alternatively, if a program has a small amount of code and data

except for one particularly large array, you could compile the program with the small or medium memory model and declare the array with the **far** or **huge** keyword.

The disadvantage of using these keywords is that they are specific to the MS-DOS implementation of Microsoft C and, thus, are not portable to other operating environments.

See Sections 6.4.1 through 6.4.4 for more information about **near**, **far**, and **huge** and for examples of their use.

8.5.2 Setting the Data Threshold

Another way to control allocation in large data models is to set a data threshold by compiling with the **/Gt** option. This option is especially useful if your program uses more than 64K of initialized static and global data and does not fit in the default data segment. Any data items larger than the value you specify are allocated to their own data segments.

8.5.3 Controlling Segments Used for Allocation

If programs compiled with large data models use external, far data items, you can tell the compiler which items reside in the same far data segment by using the **same_seg** pragma. The variables you specify in this pragma help the optimizer recognize common subexpressions involving data loads. Note that you must also compile your program with the **/ND** option to ensure that the variables you specify are allocated in the same segment.

See Section 6.7 for a description of the **/ND** option and Section 6.8 for a description of the **same_seg** pragma.

8.6 Efficiency in Large Code Models

Two linker options, **/F** and **/PAC**, can result in smaller and faster executable files and improved program-load times for programs that explicitly or implicitly use far-function calls.

The **/F** option tells the linker to optimize far calls to procedures that lie in the same segment as the caller. When you specify the **/F** option, the linker optimizes 32-bit calls to procedures in the same segment as the calling procedure. Since the segment addresses of the calling and called procedures are the same, only a 16-bit call is required. If the **/F** option is given, the linker removes the far call and replaces it with code that first places **CS** on the stack, then makes a near call. The called procedure still

returns with a far (32-bit) return instruction. However, because both the code segment (stored in **CS**) and the near address are on the stack, the far return is done correctly. The linker also adds a **NOP** instruction so that the five-byte far call is replaced by exactly five bytes of instructions.

Note

You may not want to use the **/F** option if your program includes system-level assembly-language routines or if you are linking object files that were compiled with a different C compiler. See the Microsoft CodeView and Utilities manual for more information about restrictions on the use of the **/F** option.

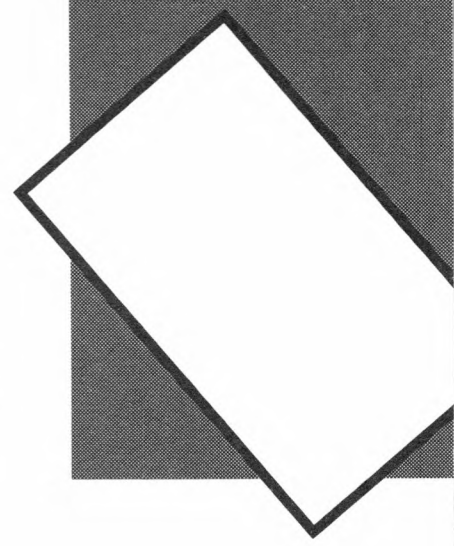
Used in conjunction with the **/F** option, the **/PAC** linker option can reduce the size and improve the efficiency of executable files. The **/PAC** option tells the linker to group neighboring code segments. Code segments in the same group share the same segment address; all offset addresses are then adjusted upward as needed. As a result, many instructions that would otherwise have different segment addresses share the same segment address.

APPENDIXES

A	Using Exit Codes	189
B	Converting from Previous Versions of the Compiler	193
C	Writing Portable Programs	209
D	Writing Programs for Read-Only Memory	227
E	Error Messages	235

APPENDIX A

USING EXIT CODES



A.1	Introduction.....	191
A.2	Exit Codes with MS-DOS Batch Files	191
A.3	Compiler Exit Codes	192

A.1 Introduction

All the programs in the Microsoft C Optimizing Compiler package return an exit code (sometimes called an “errorlevel” code) that can be used by MS-DOS batch files or other programs such as **MAKE**. If the program finishes without errors, it returns a code of 0. The code returned varies depending on the error encountered.

This appendix discusses how to use exit codes with DOS batch files and lists the exit code numbers that can be returned by the Microsoft C Optimizing Compiler. See Appendix B of the Microsoft CodeView and Utilities manual for a description of the exit code numbers returned by the other programs in the Microsoft C Optimizing Compiler package.

A.2 Exit Codes with MS-DOS Batch Files

If you use MS-DOS batch files, you can test the code returned with the **IF ERRORLEVEL** command. The sample batch file following, called **COMPILE.BAT**, illustrates how:

```
CL %1.C
IF NOT ERRORLEVEL 1 %1
```

You can execute this sample batch file with the following command:

```
COMPILE TEST
```

DOS then executes the first line of the batch file, substituting **TEST** for the parameter **%1**, as in the following command line:

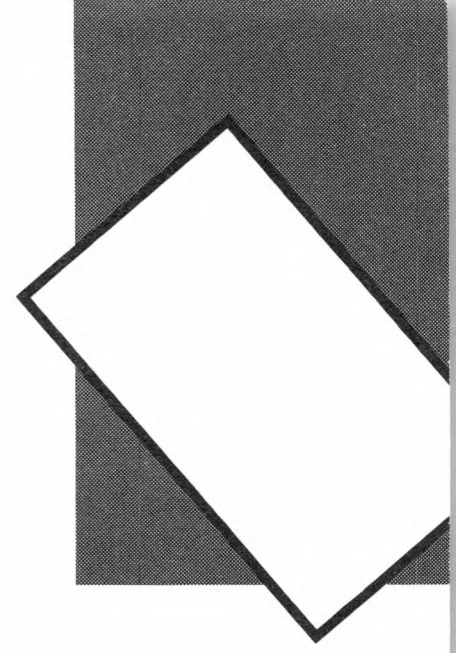
```
CL TEST.C
```

It returns a code of 0 if the compilation and linking are successful, or a higher code if an error occurs. In the second line, DOS tests to see if the code returned by the previous line is 1 or higher. If it is not (that is, if the code is 0), the **TEST** program is executed.

A.3 Compiler Exit Codes

Code	Meaning
0	No fatal error
2	Program error (such as compiler error)
4	System level error (such as out of disk space or compiler internal error)

APPENDIX B CONVERTING FROM PREVIOUS VERSIONS OF THE COMPILER



B.1	Introduction.....	195
B.2	Differences between Versions 5.0 and 4.0.....	195
B.2.1	Enhancements and Additions.....	195
B.2.2	Changes to the Language Syntax.....	196
B.2.3	New Features for the MS-DOS Implementation of C	198
B.2.4	Changed Library Routines.....	199
B.2.4.1	Graphics Routines.....	199
B.2.4.2	Heap-Checking Functions	199
B.2.4.3	DOS and BIOS Interface Functions	200
B.2.4.4	Other New Functions.....	200
B.2.4.5	New Include Files.....	201
B.3	Differences between Versions 4.0 and 3.0.....	203
B.3.1	Enhancements and Additions.....	203
B.3.2	Changes in the Language Syntax.....	204
B.3.3	New Features for the MS-DOS Implementation of C	206
B.3.4	New Library Routines and Include Files.....	207
B.3.5	Changes in Library-Function Syntax	208

B.1 Introduction

This appendix describes differences between Version 5.0 and Version 4.0, and between Version 4.0 and Version 3.0, of the Microsoft C Optimizing Compiler. If you have an earlier version of the compiler, or if you have written programs for an earlier version, this chapter can help you convert your previous source code. The actions necessary to convert source code depend on which of the earlier versions you have.

Version 5.0 is an update of Version 4.0. Generally, the two versions are compatible: most C source code written for Version 4.0 should compile without change on the Version 5.0 compiler, although there are erroneous C constructs allowed in Version 4.0 that are not allowed in Version 5.0, and changes in the emerging ANSI C standard may force changes in source programs (for more information, see the *Microsoft C Language Reference*). In some cases you may be able to enhance your programs by revising them to take advantage of new library functions and other features available with Version 5.0.

B.2 Differences between Versions 5.0 and 4.0

Changes in Version 5.0 since Version 4.0 fall into the following categories:

- Enhancements and additions to the compiler software to allow for more flexible programming, improved code generation, and increased support for the developing ANSI standard
- Changes in the language syntax
- New language features specific to the MS-DOS implementation
- New library functions and include files
- Changes in function operations, primarily to conform to the specifications for these functions in the the ANSI C standard

These features and the changes required to take advantage of them are discussed in the following sections.

B.2.1 Enhancements and Additions

Enhancements for Version 5.0 include the following:

- Improved code generation, including loop optimization; improved large-model code generation; and intrinsic functions

- Faster compilation speed
- Batch files to assist in installation of the compiler software on hard-disk systems
- Support for code that will be loaded into read-only memory (ROM)
- New error-message numbering
- Inclusion of the Microsoft QuickC™ Compiler, which comprises integrated editor, compiler, and debugger; multiple-module, in-memory compilation; and in-memory **MAKE** facility

B.2.2 Changes to the Language Syntax

Some Version 5.0 changes were made to the C language syntax to make it conform more closely to the new ANSI standard. Most of these changes do not affect source code written for the Version 4.0 compiler. The changes are summarized below:

- Full function prototyping is supported in Version 5.0. A function prototype is a forward declaration containing the types and, optionally, names of the parameters (if any) expected in the function call. It can also include identifiers for the arguments, though they go out of scope at the end of the prototype. Prototypes allow the compiler to perform type checking on the actual arguments passed when the function is called. If the compiler does not find a prototype, the first occurrence of the function (definition or call) is used as the basis of a prototype for that function. That prototype is used to perform type checking against subsequent calls, subsequent declarations, or the definition. See Chapters 4 and 7 of the *Microsoft C Language Reference* for more information about function prototyping.
- The **const** and **volatile** type specifiers have been implemented for Version 5.0. The **const** type specifier declares an object as an unmodifiable lvalue. It can be used for objects of any fundamental or aggregate type or for pointers to objects of any type. The **volatile** type specifier is implemented syntactically, but not semantically. See Chapter 4 of the *Microsoft C Language Reference* for more information.

Note

Programs that currently use **const** or **volatile** as identifiers must be recoded to use other names.

- In Version 5.0, variables of **enum** type are treated as if they were of **int** type in all cases. Therefore, **enum** variables can be used in indexing expressions and as operands of all relational and arithmetic operators.
- String concatenation is supported in Version 5.0. This feature causes adjacent string literals to be concatenated into a single string literal. This means, for example, that instead of using a backslash before a new-line character to indicate continuation of a long string literal, the literal can simply be broken into two or more quoted string literals on separate lines. See Chapter 2 of the *Microsoft C Language Reference* for more information.
- New preprocessor features in Version 5.0 include the “stringizing” operator (**#**), which allows arguments in macro expansions to be expanded into a string literal containing the expanded argument; and the “token pasting” operator (**##**), which concatenates the tokens on either side of the operator into a new token in macro expansions. See Chapter 8 of the *Microsoft C Language Reference* for more information.

Note

Previous versions of Microsoft C allowed expansion of macro formal arguments appearing in string literals and character constants. Programs that rely on this feature *must* be recoded to use the stringizing operator. See the discussion of string literals in Chapter 2 of the *Microsoft C Language Reference* for more information.

- The **long double** data type is now supported; the **long float** data type is no longer supported.
- The three-digit forms of hex escape sequences (**\xddd**) and octal escape sequences (**\ddd**) are now supported.
- The unary plus (**+**) operator is allowed, but ignored semantically.

B.2.3 New Features for the MS-DOS Implementation of C

The following new **CL** command options have been added to the MS-DOS implementation of the Microsoft C Optimizing Compiler for Version 5.0:

Option	Effect
/Oi	Enables intrinsic code generation for all available functions
/Ol	Enables loop optimizations for an entire program
/Op	Forces consistent precision in floating-point math operations
/qc	Invokes the Microsoft QuickC Compiler for fast compilation
/Sl	Specifies the line width for source listings
/Sp	Specifies the number of lines per page for source listings
/Ss	Specifies subtitles for source listings
/St	Specifies titles for source listings
/Tc	Tells the compiler that the following file is a C source file
/Zp	Packs structures on one-, two-, or four-byte boundaries

The following new pragmas have been added to the MS-DOS implementation of the Microsoft C Optimizing Compiler for Version 5.0 to control the specified features on a local basis:

Pragma	Effect
loop_opt	Turns loop optimizations on and off
pack	Specifies packing alignment for structures
intrinsic	Specifies which functions are compiled as intrinsic functions
function	Specifies which functions are compiled as standard function calls
same_seg	Tells the compiler to assume that specified variables are allocated in the same far data segment
alloc_text	Specifies modules to be grouped into a specified far code segment

Note that the existing **check_stack** pragma uses the following new format for specifying arguments:

```
#pragma check_stack([[{ on|off} ]])
```

B.2.4 Changed Library Routines

The run-time library routines provided with Version 5.0 of the Microsoft C Optimizing Compiler are moving to support the the ANSI C standard. In addition, many new functions and two new include files have been added to the library.

Sections B.2.4.1–B.2.4.5 list the new functions by type. Section B.2.4.6 describes the new include files.

B.2.4.1 Graphics Routines

The following graphics functions have been added. These functions are included in the the **GRAPHICS.LIB** library; they may also be included in the combined libraries built by the **SETUP** program. Required structures and constants for these routines are defined in the new **graph.h** include file.

_arc	_gettextcolor	_setbkcolor
_clearscreen	_gettextposition	_setcliprgn
_displaycursor	_getvideoconfig	_setcolor
_ellipse	_imagesize	_setfillmask
_floodfill	_lineto	_setlinestyle
_getbkcolor	_moveto	_setlogorg
_getcolor	_outtext	_setpixel
_getcurrentposition	_pie	_settextcolor
_getfillmask	_putimage	_settextposition
_getimage	_rectangle	_settextwindow
_getlinestyle	_remapallpalette	_setvideomode
_getlogcoord	_remappalette	_setviewport
_getphyscoord	_selectpalette	_setvisualpage
_getpixel	_setactivepage	_wrapon

B.2.4.2 Heap-Checking Functions

The following routines have been added to help debug heap-related problems in programs. These routines are defined in the **malloc.h** include file.

<code>_fheapchk</code>	<code>_heapset</code>	<code>_nheapchk</code>
<code>_fheapset</code>	<code>_heapwalk</code>	<code>_nheapset</code>
<code>_fheapwalk</code>	<code>_memmax</code>	<code>_nheapwalk</code>
<code>_heapchk</code>		

B.2.4.3 DOS and BIOS Interface Functions

The following new functions provide access to DOS system calls. Required definitions for these functions are given in the **dos.h** include file.

<code>_chain_intr</code>	<code>_dos_getdrive</code>	<code>_dos_setfileattr</code>
<code>_disable</code>	<code>_dos_getfileattr</code>	<code>_dos_setftime</code>
<code>_dos_allocmem</code>	<code>_dos_getftime</code>	<code>_dos_settime</code>
<code>_dos_close</code>	<code>_dos_gettime</code>	<code>_dos_setvect</code>
<code>_dos_creat</code>	<code>_dos_getvect</code>	<code>_dos_write</code>
<code>_dos_creatnew</code>	<code>_dos_keep</code>	<code>_enable</code>
<code>_dos_findfirst</code>	<code>_dos_open</code>	<code>_farjmp</code>
<code>_dos_findnext</code>	<code>_dos_read</code>	<code>_harderr</code>
<code>_dos_freemem</code>	<code>_dos_setblock</code>	<code>_hardresume</code>
<code>_dos_getdate</code>	<code>_dos_setdate</code>	<code>_hardretn</code>
<code>_dos_getdiskfree</code>	<code>_dos_setdrive</code>	

The following new functions provide access to ROM-BIOS interrupts. Required definitions for these functions are given in the new **bios.h** include file.

`_bios_serialcom`
`_bios_disk`
`_bios_equiplist`
`_bios_keybrd`
`_bios_memsize`
`_bios_printer`
`_bios_timeofday`

B.2.4.4 Other New Functions

Other new library functions provided with Version 5.0 are listed below:

<code>clock</code>	<code>inpw</code>	<code>memmove</code>	<code>_searchenv</code>
<code>div</code>	<code>ldiv</code>	<code>mktime</code>	<code>_splitpath</code>
<code>fgetpos</code>	<code>_lrotl</code>	<code>outpw</code>	<code>_strdate</code>
<code>fsetpos</code>	<code>_lrotr</code>	<code>_rotl</code>	<code>_strtime</code>
<code>_getdate</code>	<code>_makepath</code>	<code>_rotr</code>	<code>strtoul</code>

B.2.4.5 New Include Files

The new include files provided with Version 5.0 of the Microsoft C Optimizing Compiler are described below.

File	Purpose
bios.h	Defines the new BIOS-interface routines and the constants and structures used with these routines
graph.h	Defines the new graphics routines and the constants and structures used with these routines

For conformance with the the ANSI C standard, the following constants defined in the include file **float.h** refer to a base-2 exponent in Version 5.0:

DBL_MIN_EXP
DBL_MAX_EXP
FLT_MIN_EXP
FLT_MAX_EXP
LDBL_MIN_EXP
LDBL_MAX_EXP

In Version 4.0, these constants refer to a base-10 exponent. The base-10 versions of these constants are now named **DBL_MIN_10_EXP**, **DBL_MAX_10_EXP**, and so on.

The following table lists the existing library functions that have been changed for compatibility with the the ANSI C standard in Version 5.0 and the changes that have been made to each function:

Function	Changes
abort	Now calls raise(SIGABRT) instead of the exit function.
assert	Now calls the abort function instead of the exit function. The output from a failed assertion now contains the text of the failed expression.
calloc	Now returns NULL for calloc(0) instead of allocating a zero-length item on the heap.
cputs	Now always returns 0; no error code is returned.
ctime	Now returns NULL instead of January 1, 1980, for time values prior to January 1, 1980.

fclose and fcloseall	Now delete the specified file or files if the files were created by the tmpfile function.
gmtime	Now returns NULL instead of January 1, 1980, for values prior to January 1, 1980.
localtime	Now returns NULL instead of January 1, 1980, for values prior to January 1, 1980.
log and log10	Now set errno to ERANGE rather than to EDOM when an error occurs. Although this value is different from the value returned by the XENIX version of these functions, it is compatible with the the ANSI C standard.
malloc	Now returns NULL for malloc(0) instead of allocating a zero-length item on the heap.
onexit	Has been duplicated under the new ANSI-compatible name atexit .
memcpy	If some regions of the source and destination overlap, memcpy no longer ensures that the original source bytes in the overlapping region are copied before being overwritten. Use memmove to handle overlapping regions.
printf family	The cprintf , fprintf , and sprintf functions support the L format modifier and handle negative values for precision and field-width arguments. Also, when errors occur, these functions return -1 instead of the number of characters printed up to the point of the error.
putch	The putch function no longer returns an error code.
scanf family	The cscanf , fscanf , and sscanf functions support the L format modifier and the g , E , and G format specifiers.
setvbuf	Now uses an allocated buffer if a NULL is passed as the buffer pointer and the buffer type is _IOFBF (full buffering) or _IOLBF (line buffering). The file is unbuffered only if _IONBF is specified.
strerror	Has been renamed _strerror . The ANSI strerror function, which maps a specified error number to the corresponding error message, is also implemented in Version 5.0.

system	For a NULL pointer argument, now returns 0 and sets ERRNO to ENOENT if no COMMAND.COM file is found, or returns 1 if a COMMAND.COM file is found.
tmpfile	Now opens the temporary file in binary mode for updating (wb+) rather than default mode for updating (w+).

For more information about the new library functions, see the *Microsoft C Run-Time Library Reference*.

B.3 Differences between Versions 4.0 and 3.0

Changes between Versions 4.0 and 3.0 fall into the same categories as those between Versions 5.0 and 4.0.

- Enhancements and additions to the compiler software to allow for more flexible programming, improved code generation, and increased support for the developing ANSI standard
- Changes in the language syntax
- New language features specific to the MS-DOS implementation
- New library functions and include files

These features and the changes required to take advantage of them are discussed in the following sections.

B.3.1 Enhancements and Additions

Enhancements for Version 4.0 include the following:

- New options for **CL** and **LINK**
- Improved code optimization
- New memory models (**compact** and **huge**)
- Source listings
- Numbered error messages
- Huge arrays, allowing a single array to be larger than 64K
- Three new utilities: **MAKE**, **SETENV**, and the Microsoft Code-View symbolic debugger

These changes should have no effect on Version 3.0 source code, but you may need to revise existing batch files or **MAKE** description files to allow them to work correctly with Version 4.0.

See Chapter 3, "Compiling with the CL Command," for information on changes to the syntax of the **CL** command line.

B.3.2 Changes in the Language Syntax

Some Version 4.0 changes were made to the C language syntax to make it conform more closely to the new ANSI standard. Most of these changes do not affect source code written for the Version 3.0 compiler. The changes are summarized below:

- The **\a** escape sequence represents the bell (or alert) character in Version 4.0.

You can make your source code more portable by using **\a** instead of **\x7**. See Section 2.2.4, "Escape Sequences," of the *Microsoft C Language Reference*.

- The **signed** keyword was added.

The **signed** keyword can be used to specify signed items. This keyword is particularly useful for declaring signed **char** types in programs compiled with the **/J** option. (**/J** changes the default mode for the **char** type to unsigned.) See Section 4.2, "Type Specifiers," of the *Microsoft C Language Reference*.

- The syntax was changed for making function calls with a variable number of arguments.

The following two declarations contrast the Version 3.0 form and the Version 4.0 form:

```
int func (int,);          /* Forward declaration in
                          ** Version 3.0 syntax
                          */

int func (int,...);       /* Forward declaration in
                          ** Version 4.0 syntax
                          */
```

This change was made to conform to changes in the ANSI standard for the C language. Both forms are supported in Version 4.0 of the Microsoft C Compiler. Microsoft recommends the use of the Version 4.0 form in all programs.

- Prior to Version 4.0, the compiler allowed arbitrary strings of characters after a syntactically correct preprocessor command. To conform to the new ANSI standard, this was disallowed in Version 4.0.

Beginning with Version 4.0, the following usage, for example, causes the compiler to generate a warning message:

```
#endif      Block ends here
```

In Versions 4.0 and later, such strings must be enclosed in comment delimiters, as in the following example:

```
#endif      /* Block ends here */
```

- Names of types defined with **typedef** are not keywords in Version 4.0, as they were in Version 3.0. In Version 4.0, these names are in the same naming class as names of functions and variables, and can be redefined in a nested block.

See Section 3.6, "Naming Classes," in the *Microsoft C Language Reference*.

- Beginning with Version 4.0, the **#pragma** directive is supported. A "pragma" is an instruction to the compiler. Its syntax is similar to the syntax of preprocessor directives, but its purpose is different. The syntax is as follows:

```
# pragma charstring
```

The only pragma instruction supported in the Microsoft C Compiler, Version 4.0, is the **check_stack** pragma. This pragma is specific to MS-DOS, and is discussed in greater detail in Section 3.3.13.2, "Removing Stack Probes."

- Hexadecimal and octal integer constants are handled differently in Version 4.0 than they are in Version 3.0.

See Section 2.3, "Constants," of the *Microsoft C Language Reference* for more information.

- The extended keywords **fortran**, **pascal**, **cdecl**, **far**, **near**, and **huge** are enabled by default in Version 4.0. They can be disabled by giving the **/Za** option on the command line.
- Two new reserved words, **const** and **volatile**, were added but not implemented for Version 4.0.
- In Version 3.0, when a near pointer is converted to type **long int**, it is first converted to type **short int**, then to **long int**; as a result,

in Version 3.0 the expression in the **if** statement evaluates as true in the following fragment:

```
char *ptr = NULL;
long i;

i = (long) ptr;
if (i == 0L) {
    .
    .
    .
}
```

In Version 4.0, the conversion order of near pointers to long integers was changed so that it conforms to the order in which the compiler does all other conversions that increase the length of a variable: first the size, then the mode. (For example, the compiler converts a variable with type **char** to type **unsigned long** by first converting it to **signed long**, then to **unsigned long**.) Because of this change, the preceding code now converts `ptr` to a far pointer by loading the appropriate segment register value, then changing that to a long integer. The expression following the **if** statement would most likely be false in Version 4.0, since the segment registers do not usually contain 0.

B.3.3 New Features for the MS-DOS Implementation of C

The following features were added to the MS-DOS implementation of the C compiler for Version 4.0:

- Two new memory models: huge and compact
- The **huge**, **signed**, and **cdecl** keywords
- A pragma (**check_stack**) to control stack checking
- The **/J** option to change the default mode for the **char** type to unsigned
- The **/Gc** option to specify the alternative calling sequence and naming conventions used in Microsoft Pascal and Microsoft FORTRAN

These features are discussed in Chapter 6, "Working with Memory Models." In most cases, they will not affect existing Version 3.0 source code. However, you may be able to improve your existing programs by modifying them to take advantage of the new memory models or the **huge** keyword.

B.3.4 New Library Routines and Include Files

New library functions and include files were added to Version 4.0 of the Microsoft C Optimizing Compiler. In some cases you may wish to modify existing source code to take advantage of new library functions and include files. The new library functions are listed below:

alloca	fmsbintoieee	_nmalloc	strnicmp
_clear87	_fmsize	_nmsize	strstr
_control87	_fpreset	onexit	strtod
diecetomsbin	_freect	remove	strtol
difftime	halloc	rmtmp	tempnam
dmsbintoieee	hfree	setvbuf	tmpfile
execlpe	lfind	spawnlpe	tmpnam
execvpe	lsearch	spawnvpe	vfprintf
_expand	_memavl	stackavail	vprintf
_ffree	memicmp	_status87	vsprintf
fiecetomsbin	_msize	strerror	
_fmalloc	_nfree	stricmp	

The new include files are listed below:

File	Purpose
float.h	Defines values used in floating-point operations
limits.h	Defines upper and lower limits for various types
stdarg.h	Defines a complete set of typedef definitions and macros that can be used to write portable programs that can handle functions with variable-length argument lists; designed to be compatible with the proposed ANSI standard for C
stddef.h	Defines standard values such as NULL and errno
varargs.h	Defines a complete set of typedef definitions and macros that can be used to write portable programs that can handle functions with variable-length argument lists; designed to be compatible with UNIX System V

For more information about the new library functions and include files, see the *Microsoft C Run-Time Library Reference*.

B.3.5 Changes in Library-Function Syntax

In order to conform to the developing ANSI standard, the order of the parameters in the **rename** function was changed for Version 4.0. The syntax for Version 3.0 is as follows:

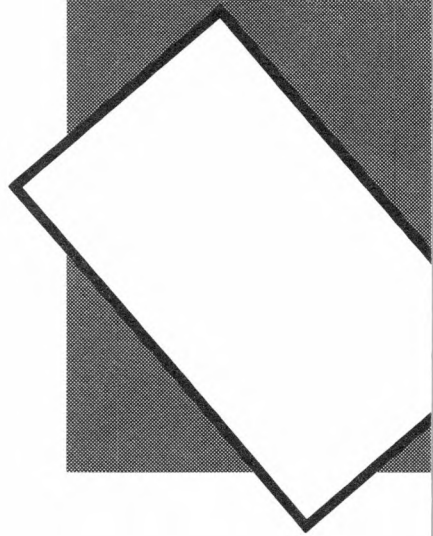
rename(*newname*, *oldname*)

The following syntax was implemented for Version 4.0:

rename(*oldname*, *newname*)

APPENDIX C

WRITING PORTABLE PROGRAMS



C.1	Introduction.....	211
C.2	Program Portability	212
C.3	Machine Hardware	212
C.3.1	Byte Length	212
C.3.2	Word Length	212
C.3.3	Storage Alignment	213
C.3.4	Byte Order in a Word.....	214
C.3.5	Bit Fields	215
C.3.6	Pointers	216
C.3.7	Address Space	217
C.3.8	Character Set	217
C.4	Compiler Differences	218
C.4.1	Signed/Unsigned char and Sign Extension	218
C.4.2	Shift Operations	218
C.4.3	Identifier Length	219
C.4.4	Register Variables	219
C.4.5	Type Conversion.....	220
C.4.6	Functions with a Variable Number of Arguments.....	221
C.4.7	Side Effects and Evaluation Order.....	221
C.5	Environment Differences	222
C.6	Portability of Data.....	223
C.7	Type-Size Summary	223
C.8	Byte-Ordering Summary	225

C.1 Introduction

The standard definition of the C programming language leaves many details to be decided in specific implementations of the language. These unspecified features of the language detract from its portability and must be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences either in target-machine hardware or in compilers. C was designed to compile efficient code for the target machine (initially a Digital Equipment Corporation PDP-11®), so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This appendix highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment. The environment is determined by the system calls and library routines a program uses during execution, file path names it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, from small eight-bit microprocessors to large mainframes. This appendix is concerned with the portability of C code in the MS-DOS and XENIX programming environments. This is a more restricted problem to consider, since all MS-DOS and XENIX operating systems to date run on hardware with the following basic characteristics:

- ASCII character set
- Eight-bit bytes
- Two-byte or four-byte integers
- Two's-complement arithmetic

These features are not formally defined for the language and may not be found in all implementations of C. However, the remainder of this appendix is devoted to those systems where these basic assumptions hold.

The C language definition contains no specification of how input and output are performed. These specifications are left to system calls and library routines on individual systems. Within XENIX systems there are system calls and library routines that can be considered portable. This version of the Microsoft C Optimizing Compiler includes system calls and library routines that can be considered portable across XENIX and MS-DOS systems. The run-time library for the Microsoft C Optimizing Compiler for MS-DOS is composed primarily of XENIX-compatible routines. By restricting the use of XENIX routines to those included in the MS-DOS library, the XENIX programmer can develop MS-DOS programs in the

XENIX environment; C programs written on MS-DOS are easily portable to XENIX.

C.2 Program Portability

A program is “portable” if it can be compiled and run successfully on different machines without alteration. There are many ways to write portable programs. One way is to avoid using inherently nonportable language features. Another is to isolate any nonportable interactions with the environment, such as I/O to nonstandard devices. For example, programs should avoid hard-coded path names unless a path name is common to all systems.

Files required at compile time (such as include files) may also introduce nonportability if the path names used are not the same on all machines. In some cases, include files containing machine-specific definitions can be used to make the source code itself portable.

C.3 Machine Hardware

Differences in the hardware of the various target machines and differences in the corresponding C compilers cause the greatest number of portability problems. This section lists problems commonly encountered.

C.3.1 Byte Length

By definition, the **char** data type in C must be large enough to hold as positive integers all members of a machine's character set. For the machines described in this appendix, the **char** size is an eight-bit byte.

C.3.2 Word Length

The size of the basic data types for a given implementation are not formally defined in the C language. Therefore, they often follow the most natural size for the underlying machine. It is safe to assume that **short** is no longer than **long**. Beyond that, no assumptions are portable. For example, on some machines **short** is the same length as **int**, whereas on others **long** is the same length as **int**.

Two areas where different **int** sizes affect program portability are the following:

1. Array indexing. For very large arrays, a variable of type **int** may not be long enough to store the indices of the highest-numbered array elements.
2. Pointer subtraction. On some machines, an **int** variable may not be long enough to store the results of pointer subtraction. See Section C.3.6, "Pointers," for more information about this problem.

Programs that need to assume the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example, the maximum positive integer (on a two's-complement machine) can be obtained with the following directive:

```
#define MAXPOS    ((int) (((unsigned) - 1) >> 1))
```

This is preferable to the following code:

```
#ifdef PDP11
#define MAXPOS 32767
#else
.
.
.
#endif
```

To find the number of bytes in an **int**, use **sizeof(int)** rather than 2, 4, or some other nonportable constant.

C.3.3 Storage Alignment

The C language defines no particular layout for storage of data items relative to each other. The layout for storage of structure elements, or unions within the structure or union, is also left undefined by the language.

Some processors require that data types longer than one byte be aligned on even-byte address boundaries. Others, such as the 8086/8088, have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays, and long words on even addresses or on even long-word addresses. Therefore, the following code sequence may give different results, depending on specific alignment requirements on different machines:


```
struct stag {
    char c;
    int i;
};
printf("%d\n", sizeof(struct stag));
```

This variation in data storage has two major implications: data accessed as nonprimitive data types are not portable; and code that makes assumptions about the layout on a particular machine is not portable.

Therefore, unions containing structures are nonportable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used exclusively to store different data in the same space at different times. For example, if the following union were used to obtain four bytes from a long word, the code would not be portable:

```
union {
    char c[4];
    long lw;
} u;
```

The **sizeof** operator should always be used when reading and writing structures, as follows:

```
struct s_tag st;
.
.
.
write(fd, &st, sizeof(st));
```

Using the **sizeof** operator ensures portability of the source code, but does not produce a portable data file. Portability of data is discussed in Section C.6.

C.3.4 Byte Order in a Word

The variation in byte order in a word affects the portability of data more than the portability of source code. However, any program that makes use of knowledge of the internal byte order in a word is not portable. For example, on some XENIX systems there is an include file **misc.h** that contains the following structure declaration:

```

/*
 * structure to access an
 * integer in bytes
 */
struct {
    char  lobyte;
    char  hibyte;
};

```

With certain less-restrictive compilers, this declaration could be used to access the high- and low-order bytes of an integer separately and in a completely nonportable way. The correct way to do this is to use mask and shift operations to extract the required byte, as shown below:

```

#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)

```

These definitions provide a portable way to extract the least-significant and the next-least-significant bytes of an integer. Since the `int` type can be either two or four bytes, depending on the machine, even these definitions do not provide a completely portable way to access the bytes of an `int`.

One result of the byte-ordering problem is that the following code sequence will not always perform as intended:

```

int c = 0;

read(fd, &c, 1);

```

On machines where the low-order byte is stored first, the value of `c` is the byte value read. On other machines, the byte is read into some byte other than the low-order one, so the value of `c` is different.

C.3.5 Bit Fields

Bit fields are not implemented in all C compilers. The Microsoft C Optimizing Compiler implements bit fields and allows them to have any length up to the size of a `long`. However, in many implementations no bit field may be larger than an `int`, and no bit field can overlap an `int` boundary. If necessary, the compiler will leave gaps and move to the next `int` boundary. To ensure portability no individual field should exceed 16 bits.

The C language makes no guarantees about whether bit fields are assigned left to right or right to left. Therefore, although bit fields may be useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely nonportable.

C.3.6 Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers do not generate warnings for nonportable pointer operations. A common nonportable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This practice usually makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. In the following code, the byte order in the array `c` is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

Code like this is usually unnecessary or invalid. It is acceptable, however, when the **malloc** function is used to allocate space for variables that do not have **char** type. The routine is declared as type **char ***, and the return value is cast to the type to be stored in the allocated memory. If this type is not **char ***, then a compiler may issue a warning concerning illegal type conversion. In addition, the **malloc** function is designed always to return a starting address suitable for storing all types of data. A compiler may not know this, so it may give an additional warning about possible data-alignment problems. In the following example, **malloc** is used to obtain memory for an array of 50 integers:

```
extern char *malloc( );
int *ip;

ip = (int *)malloc(50);
```

This example will elicit a warning message from some compilers.

The *Microsoft C Quick Reference Guide* states that a pointer can be assigned (or cast) to an integer large enough to hold it. Note that the size of the **int** type depends on the given machine and implementation. This type is **long** on some machines and **short** on others. The size may also be modified by **near** and **far** declarations. In general, do not assume that the following statement is always true:

```
sizeof(char *) == sizeof(int)
```

For example, the following construction is nonportable, assuming that the function identifier `func` is not previously declared:

```
int p;
p = (char *)func( );
```

This example assumes that a **char** pointer has the same length as an **int**.

Another consequence of different-sized **int** types on different machines is that pointer subtraction may not give the expected results. As an example of this case, subtracting pointers to the beginning and end of a very large array may give a result that is too large to store in an **int** variable, as shown in the following example:

```
int arr[20000], *b = arr, *e = &arr[20000];
int diff;
diff = e - b;    /* result too large to store in
                  int variable diff */
```

To correct this problem, coerce the result of the pointer subtraction **long** type, then assign the result to a variable of **unsigned int** type, as shown in the following example:

```
unsigned int udiff;
udiff = (long) ((int huge *)e - (int huge *)b);
```

In most implementations, the null pointer value **NULL** is defined to be the **int** value 0. The length of the 0 value can lead to problems for functions that expect pointer arguments longer than an **int**. For portable code, always use the following form to pass a **NULL** value of the correct size:

```
func( (char *)NULL );
```

C.3.7 Address Space

The address space available to a program varies considerably from system to system. Some small processors allow only 64K for program text and data combined. Others allow up to 64K of data and 64K of program text. Larger machines may allow considerably more text and possibly more data as well.

Large programs, or programs that require large data areas, may have portability problems on small machines.

C.3.8 Character Set

The C language does not require the use of the ASCII character set. In fact, the only character-set requirements are that all characters must fit in the **char** data type, and all characters must have positive values.

In the ASCII character set, all characters have values between 0 and 127 and therefore can be represented in seven bits. On an eight-bits-per-byte machine they are all positive, regardless of whether **char** is treated as signed or unsigned.

A set of character-classification macros is included as part of the run-time library for the Microsoft C Optimizing Compiler. These macros should be used for most tests on character quantities. The macros are defined in the include file **ctype.h**, and described in the *Microsoft C Run-Time Library Reference*. They appear on the pages headed **isalnum-isascii** and **isctrl-isxdigit**.

The character-classification macros provide insulation from the internal structure of the character set. In addition, the names of the macros are often more meaningful than the equivalent line of code. Compare the following two lines:

```
if(isupper(c))  
  
if((c >= 'A') && (c <= 'Z'))
```

With some of the other macros, such as **isxdigit** to test for a hexadecimal digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an **if** statement.

C.4 Compiler Differences

There are a number of C compilers running under various operating systems. The main areas of differences between compilers are outlined in this section.

C.4.1 Signed/Unsigned char and Sign Extension

The current state of the signed versus unsigned **char** problem is best described as unsatisfactory. The sign-extension problem is a serious barrier to writing portable C, and the best solution at present is to write defensive code that does not rely on particular implementation features.

C.4.2 Shift Operations

The left-shift operator (**<<**) shifts its operand a number of bits left, filling vacated bits with zeros. This is called a logical shift. When the right-shift operator (**>>**) is applied to an unsigned quantity, it performs a logical-shift operation; when it is applied to a signed quantity, the vacated bits may be filled with zeros (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code that assumes a particular implementation is nonportable.

With compilers that use arithmetic right shift, it is necessary to shift and mask the appropriate number of high-order bits to avoid sign extension, as follows:

```
char c;  
c = (c >> 3) & 0x1f;
```

You can also avoid sign extension by using the divide operator (/) as follows:

```
char c;  
c = c / 8;
```

C.4.3 Identifier Length

The use of long symbols and identifier names will cause portability problems with some compilers. To avoid these problems, a program should keep the following symbols as short as possible:

- C preprocessor symbols
- C local symbols
- C external symbols

Some loaders also place restrictions on the number of unique characters in C external symbols. Symbols unique in the first six characters are unique to most C-language processors.

In some C implementations, the case of letters in identifiers is not significant.

C.4.4 Register Variables

The number and type of register variables in a function depend on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations and should not cause a portability problem. On an 8086 or 8088 processor, up to two register declarations are significant, and they must be applied to types of size **int** or smaller.

Since the compiler ignores excess variables of **register** type, the most important **register**-type variables should be declared first. In this way, register variables that the compiler ignores will be those that are the least important.

C.4.5 Type Conversion

The C language has some rules for implicit type conversion; it also allows explicit type conversions by type casting. The most common portability problem in implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type **char** is compared with an **int**.

The following example will never evaluate true on a machine that sign-extends **char** types but treats hexadecimal numbers as unsigned:

```
char c;
if (c == 0x80) {
    .
    .
    .
}
```

The following construction is also nonportable:

```
char c;
unsigned int u;
if (u == (unsigned)c) {
    .
    .
    .
}
```

Two problems can arise in the preceding example:

1. The **char** type may be considered either signed or unsigned, depending on the implementation.
2. For implementations that consider the **char** type to be signed, two different methods of carrying out the conversion are possible: the **char** value may be sign extended to **int** type first, then converted to **unsigned** type; or the **char** type may be converted to an unsigned type of the same size, then zero extended to **int** length.

The only safe comparison between **char** type and **int** is the following:

```
int c;
if (c == 'x') {
    .
    .
    .
}
```

This comparison is reliable because C guarantees all character constants to be positive.

Type conversion also occurs when arguments are passed to functions. Types **char** and **short** become **int**. Extending the **char** type can produce unexpected results. For example, the following program yields a result of -128 on some machines:

```
char c = 128;
printf("%d\n", c);
```

The unexpected negative value is produced because **c** is converted to **int** when it is passed to the **printf** function. The function itself has no knowledge of the original type of the argument and is expecting an **int**. The correct way to handle this situation is to code defensively and allow for the possibility of sign extension, as in the following example:

```
char c = 128;
printf("%d\n", c & 0xff);
```

C.4.6 Functions with a Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is also variable. In such cases the code is dependent on the size of various data types. For portability, these cases should be avoided.

C.4.7 Side Effects and Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression or arguments to a function call. Therefore, the following statement is almost never portable:

```
func(i++, i++);
```

Even the following statement is unwise if **func** is ever likely to be replaced by a macro, since the macro may use **i** more than once:

```
func(i++);
```

Certain XENIX-compatible macros commonly appear in user programs; some of these use their argument only once, and therefore can safely be

called with a side-effect argument. To determine whether a macro handles side effects correctly, examine the code for that macro to see whether or not the argument is evaluated more than once.

Operands to the following operators are guaranteed to be evaluated left to right:

, && || ? :

Note that the comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list is not guaranteed to be processed left to right. Therefore, the following declaration on an 8086 or 8088 processor, where only two register variables may be declared, could give any two of the four variables **register** type, depending on the compiler:

```
register int a, b, c, d;
```

To give register storage to the most important variables, use separate declaration statements and declare the most important variables first. The order of processing of individual declaration statements is guaranteed to be sequential in the following statements:

```
register int a;  
register int b;  
register int c;  
register int d;
```

C.5 Environment Differences

Most programs make system calls and use library routines for various services. This section indicates some of those routines that are not always portable and those that particularly aid portability.

System calls specific to an operating system are not portable if they are not present on all other operating-system implementations of C. Most of the system calls defined in the Microsoft DOS run-time library are compatible with XENIX system calls and are therefore portable to a XENIX environment.

Any program is nonportable that contains hard-coded path names to files or directories, or that contains user identifier numbers, log-in names, terminal lines or other system-dependent parameters. These types of constants should be in header files, passed as command-line arguments, or obtained from the environment.

Note that the members of the **printf** and **scanf** families of functions, including **fprintf**, **fscanf**, **printf**, **sprintf**, **scanf**, **vfprintf**, **vprintf**, **vsprintf**, and **sscanf**, have evolved in several ways, and some features are not completely portable. Some of the format-conversion characters have changed their meanings, in particular those relating to uppercase and lowercase in the output of hexadecimal numbers and the specification of **long** integers on 16-bit word machines. The Microsoft C specifications for these routines are given in the *Microsoft C Run-Time Library Reference*.

The names of code-helper functions (for example, **--almul**) have been changed between the MS-DOS and XENIX versions of Microsoft C. As a result, users who port object files compiled under Version 4.0 or later of the MS-DOS C compiler must move copies of the relevant helper functions from the standard combined C library (or from **LIBH.LIB** if they are using uncombined libraries) to XENIX, since Version 2.2 of the XENIX C compiler and cross-development libraries are from a different version.

Users should beware of porting object files that reference the **setjmp** or **longjmp** functions from XENIX to MS-DOS, unless these object files were compiled with the **-dos** option. The MS-DOS versions of these functions use a larger buffer size and may cause memory to be overwritten. Such object files can be ported from MS-DOS to XENIX without problems, and the corresponding source files can be ported in either direction.

C.6 Portability of Data

Data files are almost always nonportable across different central-processing-unit (CPU) architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way to achieve data-file portability is to write and read data files as one-dimensional character arrays. This procedure prevents alignment and padding problems if the data are written and read as characters, and interpreted that way. Thus ASCII text files can usually be moved between different machine types without significant problems.

C.7 Type-Size Summary

Table C.1 summarizes the sizes of the various data types as defined in the Microsoft C Optimizing Compiler, Version 5.0.

Table C.1
C Type Sizes

Type Name	Alternative Name	Storage	Range of Values
char	signed char	1 byte	–128 to 127
int	signed signed int	Implementation dependent (2 bytes in Microsoft C 5.0)	(–32,768 to 32,767 for Microsoft C Version 5.0)
short	short int signed short signed short int	2 bytes	–32,768 to 32,767
long	long int signed long signed long int	4 bytes	–2,147,483,648 to 2,147,483,647
unsigned¹ char	none	1 byte	0 to 255
unsigned	unsigned int	Implementation dependent (2 bytes in Microsoft C 5.0)	(0 to 65,535 for Microsoft C 5.0)
unsigned short	unsigned short int	2 bytes	0 to 65,535
unsigned long	unsigned long int	4 bytes	0 to 4,294,967,295
enum	none	Implementation dependent (2 bytes in Microsoft C 5.0)	(0 to 65,535 for Microsoft C 5.0)
float	none	4 bytes	Approximately 3.4E–38 to 3.4E+38 (7-digit precision)
double	none	8 bytes	Approximately 1.7E–308 to 1.7E+308 (15-digit precision)
long double	none	Implementation dependent (8 bytes in Microsoft C 5.0)	Approximately 1.7E–308 to 1.7E+308 (15-digit precision)

¹ Any type size modified by the **unsigned** keyword can be modified by the **signed** keyword instead. The **signed** keyword is useful if the **/J** option has been used to change the default sign of the **char** type.

C.8 Byte-Ordering Summary

Tables C.2 and C.3 summarize byte ordering for **short** and **long** types, respectively. The following conventions are used in these tables:

1. The lowest physically addressed byte of the data item is **a0**; **a1** has the byte address **a0 + 1**, and so on.
2. The least-significant byte of the data item is **b0**; **b1** is the next least significant, and so on.

Since byte ordering is machine specific, any program that actually makes use of the following information is guaranteed to be nonportable:

Table C.2
Byte Ordering for Short Types

CPU	Byte Order
	a0 a1
8086	b0 b1
80286	b0 b1
PDP-11®	b0 b1
VAX-11®	b0 b1
M68000	b1 b0
Z8000®	b1 b0

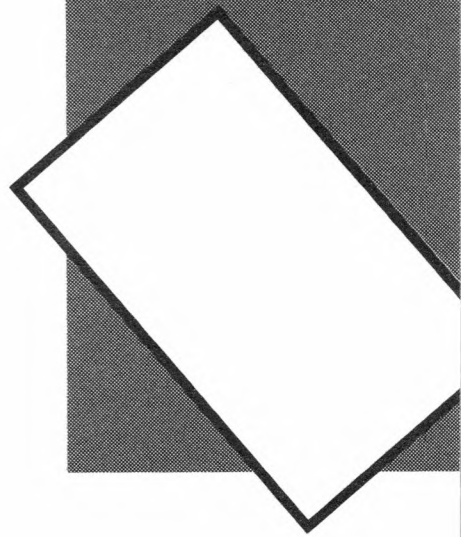
Table C.3
Byte Ordering for Long Types

CPU	Byte Order
	a0 a1 a2 a3
8086	b0 b1 b2 b3
80286	b0 b1 b2 b3
PDP-11	b2 b3 b0 b1
VAX-11	b0 b1 b2 b3
M68000	b3 b2 b1 b0
Z8000	b3 b2 b1 b0

APPENDIX D

WRITING PROGRAMS

FOR READ-ONLY MEMORY



D.1	Introduction	229
D.2	MS-DOS-Dependent Library Routines	229
D.3	Floating-Point Math Support.....	230
D.4	Modifying Start-Up Code.....	231

D.1 Introduction

This appendix presents information for developers who will be downloading code written with the Microsoft C Optimizing Compiler into read-only memory (ROM). Code of this type is more commonly known as “ROM-able” code. Information is given about the following topics:

- Run-time library routines that directly interface with MS-DOS (Section D.2)
- Floating-point math in ROMable code (Section D.3)
- Changing start-up code for non-MS-DOS environments (Section D.4)

D.2 MS-DOS-Dependent Library Routines

Because ROMable programs are often run outside an MS-DOS environment, they cannot include calls to run-time library routines that perform their operations through calls to MS-DOS functions. Table D.1 lists the library routines that call MS-DOS functions.

Table D.1

MS-DOS-Dependent Library Routines

abort	execvp	fstat	mkdir	spawnvp
access	execvpe	ftell	mktemp	spawnvpe
chdir	_exit	ftime	open	sprintf
chmod	fclose	fwrite	perror	sscanf
chsize	fcloseall	getch	printf	stat
close	fgetc	getche	putch	system
cprintf	fgetchar	getcwd	puts	tell
cputs	fgets	getpid	putw	tempnam
creat	filelength	gets	read	time
cscanf	flush	getw	remove	tmpfile
dos	flushall	halloc	rename	tmpnam
dosexterr	_fmalloc	hfree	rmdir	unlink
dup	fopen	int86	rmtmp	utime
dup2	fprintf	int86x	scanf	vfprintf
eof	fputc	intdos	sopen	vprintf
execl	fputchar	intdosx	spawnl	vsprintf
execle	fputs	kbhit	spawnle	write
execlp	fread	labs	spawnlp	
execlpe	freopen	localtime	spawnlpe	
execv	fscanf	locking	spawnv	
execve	fseek	lseek	spawnve	

A program containing calls to any these routines cannot run in a non-MS-DOS environment unless you do one of the following:

- Write replacements for these MS-DOS dependent routines as needed.
- Edit the program to remove the calls to the listed routines.
- Obtain the library source files from Microsoft and edit them so that they do not include MS-DOS function calls, and write functional equivalents of the MS-DOS functions that can be called from your program.

Note that certain functions that are not listed above may call MS-DOS functions indirectly: that is, they may be part of a series of nested calls that call routines in the list.

You may want to try to delete certain MS-DOS-dependent object modules from the C run-time library by using the Microsoft Library Manager, **LIB**. Then, when you link your ROMable program, any unresolved references could help determine which MS-DOS dependencies still need to be eliminated before the program code is burned into ROM.

Even in an MS-DOS environment, the **exec** family of functions (MS-DOS Versions 2.x and 3.x) and the **spawn** family of functions (MS-DOS Versions 2.x) may alter the code segment. As a result, these functions would not work if all of their code was in ROM.

D.3 Floating-Point Math Support

Programs that use the various floating-point math packages (described in Chapter 7, "Controlling Floating-Point Math Operations") can be used to produce ROMable code.

Each of the three floating-point math packages contains certain error-message code that depends on MS-DOS through calls to the **write** and **--nmsg_write** functions. You can eliminate the MS-DOS dependencies by providing replacements for these routines. The **--nmsg_write** routine is provided in the file named **NMSGHDR.ASM**, which the **SETUP** program installs in your *basedir*\SRC subdirectory. The error-message code in the math packages also calls the **exit** function in the C start-up code, which is MS-DOS dependent.

Programs that use the alternate math package (that is, programs compiled with the **/FPa** option or linked explicitly with an **mLIBCA.LIB** library) should produce ROMable code easily.

In order to work in a non-MS-DOS environment, programs that use the emulator math package (that is, programs compiled with the **/FPi** or **/FPc** option or linked explicitly with **mLIBCE.LIB**) must meet one of two conditions:

1. An 8087 or 80287 coprocessor is not present.
2. The environment variable **NO87** is set to a non-null value.

However, you would have to replace some of the MS-DOS calls and other interrupts by providing your interrupt handlers for them (as described in Section E.4 below).

Programs that use the 8087/80287 math package (that is, programs compiled with the **/FPi87** or **/FPc87** option or linked explicitly with **mLIBC7.LIB**) may have problems if they are placed in ROM, since code is provided in the run-time libraries to fix up floating-point instructions at run time (that is, to change the code when the instruction is first executed). The advantage of these fixups is that they allow code linked with **mLIBCE.LIB** to be run whether or not a coprocessor is installed. Since code placed in ROM cannot modify itself, a way is needed to circumvent these fixups at run time. The **FIXUPS.OBJ** module in the run-time library must be replaced by a module that sets the following public constants (absolutes) to zero:

FIARQQ	FIERQQ	FJARQQ
FICRQQ	FISRQQ	FJCRQQ
FIDRQQ	FIWRQQ	FJSRQQ

You must provide your own replacement module for **FIXUPS.OBJ** if you decide that you want to use the coprocessor math option.

D.4 Modifying Start-Up Code

In a non-MS-DOS environment, where programs typically have no need of disk-file support, you can safely delete the file initializers and terminators from the start-up file **CRT0.ASM**. In addition, some of the code that sets up and restores interrupt vectors in this file may not be appropriate to the needs of your program. In these cases, you may want to substitute your own interrupt handlers.

The start-up code for the Microsoft C Optimizing Compiler also initializes floating-point math support for programs that use it. The exact start-up support that must be provided depends on which floating-point math option you will be using for your programs. (See Chapter 7, "Controlling Floating-Point Math Operations," for a description of the floating-point

options available with the Microsoft C Optimizing Compiler.) The following paragraphs describe the math support that is currently provided in the start-up module.

Calls to the `--fpmath` routine in the `CRT0DAT.ASM` module initialize and terminate floating-point math support and set signal addresses for all five floating-point math options. Arguments to `--fpmath` have the following effects:

- A call with an argument of 0 initializes floating-point support.
- A call with an argument of 3 sets a signal address used for floating-point errors, but does not set any interrupts.
- If floating-point support needs to be terminated, `--fpmath` is called with an argument of 2.

If you choose the alternate math package (that is, if you compile your programs with the `/FPa` option and link with one of the `mLIBCA.LIB` libraries), the floating-point initialization code (`--fpmath(0)`) simply sets up the floating-point stack.

If you choose the emulator math package (compile with `/FPi` or `/FPc` and link with `mLIBCE.LIB`) or the 8087/80287 math package (compile with `/FPi87` or `/FPc87` and link with `mLIBC7.LIB`), the initialization code (`--fpmath(0)`) sets up several interrupt vectors, including 0x34 through 0x3D for internal use by software in the run-time library. If the coprocessor is present and to be used, then the nonmaskable interrupt vector (NMI—0x02) is set to `--fpinterrupt87`, and the CTRL+C signal is dealt with (as shown in the `EMOEM.ASM` file in the `\basedir\SRC` subdirectory). All these interrupts are restored with the `--fpmath(2)` call on program termination.

Interrupt vectors are processed through calls to MS-DOS functions, using interrupt 0x21. The MS-DOS function numbers (that is, the settings in the `AH` register) are 0x25 for setting interrupts and 0x35 for getting a vector that is already set. For both the set-vector and get-vector functions, the value contained in the `AL` register indicates the interrupt-vector number.

To be able to use this code in a non-MS-DOS environment, you must replace the interrupt handler provided by MS-DOS interrupt 0x21. Another MS-DOS call that is used in the floating-point initialization (`--fpmath(0)`) call is the `DOS_getversion` call (where the `AH` register contains 0x30); see the `--FPINSTALL87` routine in the `EMOEM.ASM` file. Another possibility is to replace the `--fpmath` routine and set up the interrupts in a way that avoids MS-DOS calls.

An additional piece of initialization code in the `--fpmath(0)` routine checks the environment block to see whether the **NO87** environment variable is set. In a non-MS-DOS environment, in which you have removed the `--setargv` and `--setenvp` routines from the program start-up code, this check tells the code to assume that **NO87** is not set. Since this code is executed only if you are using the emulator math package when the coprocessor is present, it in most cases should not cause problems.

The initialization and termination calls also perform **IN** and **OUT** instructions to ports related to the 8259 interrupt controllers. See the **EMOEM.ASM** file to see whether these instructions apply to, or need to be modified for, your particular hardware configuration.

APPENDIX E

ERROR MESSAGES

E.1	Introduction.....	237
E.2	Command-Line Error Messages.....	237
E.2.1	Command-Line Fatal-Error Messages.....	238
E.2.2	Command-Line Error Messages	238
E.2.3	Command-Line Warning Messages.....	241
E.3	Compiler Error Messages.....	243
E.3.1	Fatal-Error Messages.....	244
E.3.2	Compilation-Error Messages.....	251
E.3.3	Warning Messages.....	269
E.3.4	Compiler Limits.....	280
E.4	Run-Time Error Messages.....	281
E.4.1	Run-Time-Library Error Messages	281
E.4.2	Floating-Point Exceptions	284
E.4.3	Run-Time Limits	286

E.1 Introduction

This appendix lists error messages you may encounter as you develop a program, and gives a brief description of actions you can take to correct the errors. The following list tells where to find error messages for the various components of Microsoft C:

Component	Section
The command line used to invoke the Microsoft C Optimizing Compiler	Section E.2, "Command-Line Error Messages"
Microsoft C Optimizing Compiler	Section E.3, "Compiler-Error Messages"
The Microsoft C run-time libraries and run-time situations	Section E.4, "Run-Time Error Messages"

See Section E.3.4 for information about compiler limits. See the Microsoft CodeView and Utilities manual for a list of the error messages generated by the following programs:

- The Microsoft CodeView Window-Oriented Debugger (**CV.EXE**)
- The Microsoft Overlay Linker (**LINK**)
- The Microsoft Library Manager (**LIB**)
- The Microsoft Program Maintenance Utility (**MAKE**)
- The Microsoft EXE File Compression Utility (**EXEPACK**)
- The Microsoft EXE File Header Utility (**EXEMOD**)
- The Microsoft Environment Expansion Utility (**SETENV**)
- The Microsoft Standard Error Redirection Utility (**ERROUT**)

E.2 Command-Line Error Messages

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

command line fatal error D1xxx:	<i>messagetext</i>	(fatal error)
command line error D2xxx:	<i>messagetext</i>	(error)
command line warning D4xxx:	<i>messagetext</i>	(warning error)

If possible, the compiler continues operation, printing a warning message. In some cases, command-line errors are fatal and the compiler terminates processing.

E.2.1 Command-Line Fatal-Error Messages

The following messages identify fatal errors. The compiler driver cannot recover from a fatal error; it terminates after printing the error message.

Number	Command-Line Fatal-Error Message
D1000	UNKNOWN COMMAND LINE FATAL ERROR Contact Microsoft Technical Support The compiler detected an unknown fatal-error condition. Please report this condition to Microsoft Corporation using the Product Assistance Request at the back of this manual.
D1001	could not execute ' <i>filename</i> ' The compiler could not find the given file in the current working directory or any of the other directories named in the PATH variable.
D1002	too many open files, cannot redirect ' <i>filename</i> ' No more file handles were available to redirect the output of the /P option to a file. Try editing your CONFIG.SYS file and increasing the value <i>num</i> on the line files=<i>num</i> (if <i>num</i> is less than 20).

E.2.2 Command-Line Error Messages

When the compiler driver encounters any of the errors listed in this section, it continues compiling the program (if possible) and outputs additional error messages. However, no object file is produced.

Number	Command-Line Error Message
D2000	UNKNOWN COMMAND LINE ERROR Contact Microsoft Technical Support The compiler detected an unknown error condition. Please report this condition to Microsoft Corporation using the Product Assistance Request at the back of this manual.

Number	Command-Line Error Message
D2001	<p>too many symbols predefined with <code>-D</code></p> <p>Too many symbolic constants were defined using the <code>/D</code> option on the command line.</p> <p>The limit on command-line definitions is normally 16; you can use the <code>/U</code> or <code>/u</code> option to increase the limit to 20.</p>
D2002	<p>a previously defined model specification has been overridden</p> <p>Two different memory models were specified; the model specified later on the command line was used.</p>
D2003	<p>missing source file name</p> <p>You did not give the name of the source file to be compiled.</p>
D2007	<p>bad <i>option</i> flag, would overwrite '<i>string1</i>' with '<i>string2</i>'</p> <p>The specified option was given more than once, with conflicting arguments <i>string1</i> and <i>string2</i>.</p>
D2008	<p>too many <i>option</i> flags, '<i>string</i>'</p> <p>Too many letters were given with the specified option (for example, with the <code>/O</code> option).</p>
D2009	<p>unknown option <i>character</i> in '<i>optionstring</i>'</p> <p>One of the letters in the given option was not recognized.</p>
D2010	<p>unknown floating point option</p> <p>The specified floating-point option (an <code>/FP</code> option) was not one of the valid options.</p>
D2011	<p>only one floating point model allowed</p> <p>You specified more than one floating-point (<code>/FP</code>) option on the command line.</p>
D2012	<p>too many linker flags on command line</p> <p>You tried to pass more than 128 separate options and object files to the linker.</p>

Number	Command-Line Error Message
D2013	<p>incomplete model specification</p> <p>Not enough characters were given for the <i>/Astring</i> option.</p> <p>The <i>/Astring</i> option requires all three letters (to specify the data-pointer size, code-pointer size, and segment setup, respectively).</p>
D2014	<p>-ND not allowed with -Ad</p> <p>You cannot rename the default data segment unless you give the <i>/Auxx</i> option (SS != DS, load DS) on the command line.</p>
D2015	<p>assembly files are not handled</p> <p>You gave a file name with an extension of .ASM on the command line.</p> <p>Because the compiler cannot invoke the Microsoft Macro Assembler (MASM) automatically, it cannot assemble such files.</p>
D2016	<p>-Gw and -ND <i>name</i> are incompatible</p> <p>You tried to rename the default data segment to the given name when you specified the <i>/Gw</i> option.</p> <p>Renaming the default data segment is illegal in this case because the <i>/Gw</i> option requires the <i>/Auxx</i> option.</p>
D2017	<p>-Gw and -Au flags are incompatible</p> <p>You tried to specify the <i>/Auxx</i> option (SS != DS, load DS) with the <i>/Gw</i> option.</p> <p>Specifying <i>/Auxx</i> with <i>/Gw</i> is illegal because the <i>/Gw</i> option requires the <i>/Auxx</i> option.</p>
D2018	<p>cannot open linker cmd file</p> <p>The response file used to pass object-file names and options to the linker could not be opened.</p> <p>This error may have occurred because another read-only file had the same name as the response file.</p>
D2019	<p>cannot overwrite the source file, '<i>name</i>'</p> <p>You specified the source file as an output-file name.</p> <p>The compiler does not allow the source file to be overwritten by one of the compiler output files.</p>

Number	Command-Line Error Message
D2020	<p>-Gc option requires extended keywords to be enabled (-Ze)</p> <p>The /Gc option and the /Za option were specified on the same command line.</p> <p>The /Gc option requires the extended keyword cdecl to be enabled if library functions are to be accessible.</p>
D2021	<p>invalid numerical argument '<i>string</i>'</p> <p>A non-numerical string was specified following an option that required a numerical argument.</p>
D2022	<p>cannot open help file, cl.hlp</p> <p>The /HELP option was given, but the file containing the help messages (CL.HLP) was not in the current directory or in any of the directories specified by the PATH environment variable.</p>
D2023	<p>invalid model specification - small model only</p>

E.2.3 Command-Line Warning Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking.

Number	Command-Line Warning Message
D4000	<p>UNKNOWN COMMAND LINE WARNING</p> <p>Contact Microsoft Technical Support</p> <p>An unknown fatal condition has been detected by the compiler. Please report this condition to Microsoft Corporation using the Product Assistance Request at the back of this manual.</p>
D4001	<p>listing has precedence over assembly output</p> <p>Two different listing options were chosen; the assembly listing is not created.</p>
D4002	<p>ignoring unknown flag '<i>string</i>'</p> <p>One of the options given on the command line was not recognized and is ignored.</p>

Number	Command-Line Warning Message
D4003	<p>80186/286 selected over 8086 for code generation</p> <p>Both the /G0 option and either the /G1 or /G2 option were given; /G1 or /G2 takes precedence.</p>
D4004	<p>optimizing for time over space</p> <p>This message confirms that the /Ot option is used for optimizing.</p>
D4005	<p>Please enter new file name (full path) or Ctrl+C to quit</p> <p>The CL command could not find the specified executable file in the search path.</p>
D4006	<p>only one of -P/-E/-EP allowed, -P selected</p> <p>Only one preprocessor output option can be specified at one time.</p>
D4007	<p>-C ignored (must also specify -P or -E or -EP)</p> <p>The /C option must be used in conjunction with one of the preprocessor output flags, /E, /EP, or /P.</p>
D4008	<p>non-standard model -- defaulting to small model libraries</p> <p>A nonstandard memory model was specified with the /Astring option. The library search records in the object model were set to use the small-model libraries.</p>
D4009	<p>threshold only for far/huge data, ignored</p> <p>The /Gt option cannot be used in memory models that have near data pointers. It can be used only in compact, large, and huge models.</p>
D4010	<p>-Gp not implemented, ignored</p> <p>The MS-DOS version of Microsoft C does not support profiling.</p>
D4011	<p>preprocessing overrides source listing</p> <p>Only a preprocessor listing was generated, since the compiler cannot generate both a source listing and a preprocessor listing at the same time.</p>

Number	Command-Line Warning Message
D4012	function declarations override source listing The compiler cannot generate both a source-listing file and the function prototype declarations at the same time.
D4013	combined listing has precedence over object listing When /Fc is specified along with either /Fl or /Fa , the combined listing (/Fc) is created.
D4014	invalid value <i>number</i> for ' <i>string</i> '. Default <i>number</i> is used An invalid value was given in a context where a particular numerical value was expected.
D4017	conflicting stack checking options - stack checking disabled Both the /Ge and the /Gs flags are given in one compile command (/Ge enables stack checking, /Gs disables it).

E.3 Compiler Error Messages

The error messages produced by the C compiler fall into three categories:

1. Fatal-error messages
2. Compilation-error messages
3. Warning messages

The messages for each category are listed below in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number. All messages give the file name and line number where the error occurs.

Fatal-Error Messages

Fatal-error messages indicate a severe problem, one that prevents the compiler from processing your program any further. These messages have the following format:

filename(line) : fatal error C1xxx: messagetext

After the compiler displays a fatal-error message, it terminates without producing an object file or checking for further errors.

Compilation-Error Messages

Compilation-error messages identify actual program errors. These messages appear in the following format:

filename(line) : error C2xxx: messagetext

The compiler does not produce an object file for a source file that has compilation errors in the program. When the compiler encounters such errors, it attempts to recover from the error. If possible, it continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Warning Messages

Warning messages are informational only; they do not prevent compilation and linking. These messages appear in the following format:

filename(line) : warning C4xxx: messagetext

You can use the `/W` option to control the level of warnings that the compiler generates. This option is described in Section 3.3.11.2.

E.3.1 Fatal-Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after printing the error message.

Number	Fatal-Error Message
--------	---------------------

C1000	UNKNOWN FATAL ERROR Contact Microsoft Technical Support An unknown error condition has been detected by the compiler. Please report this condition to Microsoft Corporation, using the Product Assistance Request at the back of this manual.
C1001	Internal Compiler Error Contact Microsoft Technical Support The compiler detected an internal inconsistency. Please report this condition to Microsoft Corporation using the Product Assistance Request at the back of this manual.

Number	Fatal-Error Message
	Please include the file name and line number where the error occurred in this report; note that the file name refers to an internal compiler file, <i>not</i> your source file.
C1002	<p>out of heap space</p> <p>The compiler has run out of dynamic memory space. This usually means that your program has many symbols and/or complex expressions.</p> <p>To correct the problem, divide the file into several smaller source files, or break expressions into subexpressions.</p>
C1003	<p>error count exceeds <i>n</i>; stopping compilation</p> <p>Errors in the program were too numerous or too severe to allow recovery, and the compiler must terminate.</p>
C1004	<p>unexpected EOF</p> <p>This message appears when you have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file. This message can also occur when a comment does not have a closing delimiter (<i>*/</i>), or when an #if directive occurs without a corresponding closing #endif directive.</p>
C1005	<p>string too big for buffer</p> <p>A string in a compiler intermediate file overflowed a buffer.</p>
C1006	<p>write error on compiler intermediate file</p> <p>The compiler was unable to create the intermediate files used in the compilation process.</p> <p>The following conditions commonly cause this error:</p> <ol style="list-style-type: none">1. Too few files in the files=<i>number</i> line of the CONFIG.SYS file (the compiler requires <i>number</i> to be at least 15)2. Not enough space on a device containing a compiler intermediate file
C1007	<p>unrecognized flag '<i>string</i>' in '<i>option</i>'</p> <p>The <i>string</i> in the command-line <i>option</i> was not a valid option.</p>

Number	Fatal-Error Message
C1009	<code>compiler limit</code> <code>possibly a recursively defined macro</code> The expansion of a macro exceeds the available space. Check to see if the macro is recursively defined, or if the expanded text is too large.
C1010	<code>compiler limit : macro expansion too big</code> The expansion of a macro exceeds the available space.
C1012	<code>bad parenthesis nesting - missing 'character'</code> The parentheses in a preprocessor directive were not matched; <i>character</i> is either a left or right parenthesis.
C1013	<code>cannot open source file 'filename'</code> The given file either did not exist, could not be opened, or was not found. Make sure your environment settings are valid and that you have given the correct path name for the file.
C1014	<code>too many include files</code> Nesting of #include directives exceeds 10 levels.
C1016	<code>#if[n]def expected an identifier</code> You must specify an identifier with the #ifdef and #ifndef directives.
C1017	<code>invalid integer constant expression</code> The expression in an #if directive must evaluate to a constant.
C1018	<code>unexpected '#elif'</code> The #elif directive is legal only when it appears within an #if , #ifdef , or #ifndef directive.
C1019	<code>unexpected '#else'</code> The #else directive is legal only when it appears within an #if , #ifdef , or #ifndef directive.
C1020	<code>unexpected '#endif'</code> An #endif directive appears without a matching #if , #ifdef , or #ifndef directive.

Number	Fatal-Error Message
C1021	bad preprocessor command ' <i>string</i> ' The characters following the number sign (#) do not form a valid preprocessor directive.
C1022	expected '#endif' An #if, #ifdef, or #ifndef directive was not terminated with an #endif directive.
C1026	parser stack overflow, please simplify your program Your program cannot be processed because the space required to parse the program causes a stack overflow in the compiler. To solve this problem, try to simplify your program.
C1027	DGROUP data allocation exceeds 64K More than 64K of variables was allocated to the default data segment. For compact-, medium-, large-, or huge-model programs, use the /Gt option to move items into separate segments.
C1032	cannot open object listing file ' <i>filename</i> ' One of the following statements about the file name or path name given (<i>filename</i>) is true: <ol style="list-style-type: none">1. The given name is not valid.2. The file with the given name cannot be opened for lack of space.3. A read-only file with the given name already exists.
C1033	cannot open assembly-language output file ' <i>filename</i> ' One of the conditions listed under error message C1032 prevents the given file from being opened.
C1034	cannot open source file ' <i>filename</i> ' One of the conditions listed under error message C1032 prevents the given file from being opened.

Number	Fatal-Error Message
C1035	<p>expression too complex, please simplify</p> <p>The compiler cannot generate the code for a complex expression.</p> <p>Break the expression into simpler subexpressions and recompile.</p>
C1036	<p>cannot open source listing file '<i>filename</i>'</p> <p>One of the conditions listed under error message C1032 prevents the given file from being opened.</p>
C1037	<p>cannot open object file '<i>filename</i>'</p> <p>One of the conditions listed under error message C1032 prevents the given file from being opened.</p>
C1039	<p>unrecoverable heap overflow in Pass 3</p> <p>The post-optimizer compiler pass overflowed the heap and could not continue.</p> <p>Try recompiling with the /Od option (see Section 3.3.13, "Optimizing"), or try breaking up the function containing the line that caused the error.</p>
C1040	<p>unexpected EOF in source file '<i>filename</i>'</p> <p>The compiler detected an unexpected end-of-file condition while creating a source listing or mingled source/object listing.</p> <p>This error probably occurred because the source file was edited during compilation.</p>
C1041	<p>cannot open compiler intermediate file — no more files</p> <p>The compiler could not create intermediate files used in the compilation process because no more file handles were available.</p> <p>This error can usually be corrected by changing the <code>files=<i>number</i></code> line in CONFIG.SYS to allow a larger number of open files (20 is the recommended setting).</p>
C1042	<p>cannot open compiler intermediate file — no such file or directory</p> <p>The compiler could not create intermediate files used in the compilation process because the TMP environment variable was set to an invalid directory or path.</p>

Number	Fatal-Error Message
C1043	<p>cannot open compiler intermediate file</p> <p>The compiler could not create intermediate files used in the compilation process. The exact reason is unknown.</p>
C1044	<p>out of disk space for compiler intermediate file</p> <p>The compiler could not create intermediate files used in the compilation process because no more space was available.</p> <p>To correct the problem, make more space available on the disk and recompile.</p>
C1045	<p>floating point overflow</p> <p>The compiler generated a floating-point exception while doing constant arithmetic on floating-point items at compile time, as in the following example:</p> <pre>float fp_val = 1.0e100;</pre> <p>In this example, the double-precision constant 1.0e100 exceeds the maximum allowable value for a floating-point data item.</p>
C1047	<p>too many <i>option</i> flags, '<i>string</i>'</p> <p>The <i>option</i> appeared too many times. The <i>string</i> contains the occurrence of the option that caused the error.</p>
C1048	<p>Unknown option '<i>character</i>' in '<i>optionstring</i>'</p> <p>The <i>character</i> was not a valid letter for <i>optionstring</i>.</p>
C1049	<p>invalid numerical argument '<i>string</i>'</p> <p>A numerical argument was expected instead of <i>string</i>.</p>
C1050	<p>code segment '<i>segmentname</i>' too large</p> <p>A code segment grew to within 36 bytes of 64K during compilation.</p> <p>A 36-byte pad is used because of a bug in some 80286 chips that can cause programs to exhibit strange behavior when, among other conditions, the size of a code segment is within 36 bytes of 64K.</p>

Number	Fatal-Error Message
C1052	<p>too many <code>#if/#ifdef</code>'s</p> <p>You have exceeded the maximum nesting level for <code>#if/#ifdef</code> directives.</p>
C1053	<p>compiler limit : struct/union nesting</p> <p>Structure and union definitions were nested to more than 10 levels.</p>
C1054	<p>compiler limit : initializers too deeply nested</p> <p>The compiler limit on nesting of initializers was exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized.</p> <p>To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.</p>
C1056	<p>compiler limit : out of macro expansion space</p> <p>The compiler has overflowed an internal buffer during the expansion of a macro; reduce the complexity of the macro.</p>
C1057	<p>unexpected EOF in macro expansion; (missing ') '?)</p> <p>The compiler has encountered the end of the source file while gathering the arguments of a macro invocation. Usually this is the result of a missing closing parenthesis () on the macro invocation.</p>
C1059	<p>out of near heap space</p> <p>The compiler has run out of storage for items that it stores in the "near" (default data segment) heap. This usually means that your program has too many symbols or complex expressions. To correct the problem, divide the file into several smaller source files, or break expressions into smaller subexpressions.</p>
C1060	<p>out of far heap space</p> <p>The compiler has run out of storage for items that it stores in the "far" heap. Usually this is the result of too many symbols in the symbol table.</p>

E.3.2 Compilation-Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues parsing the program (if possible) and outputs additional error messages. However, no object file is produced.

Number	Compilation-Error Message
C2000	UNKNOWN ERROR Contact Microsoft Technical Support The compiler detected an unknown error condition. Please report this condition to Microsoft Corporation using the Product Assistance Request at the back of this manual.
C2001	<code>newline in constant</code> A new-line character in a character or string constant was not in the correct escape-sequence format (<code>\n</code>).
C2002	<code>out of macro actual parameter space</code> Arguments to preprocessor macros exceeded 256 bytes.
C2003	<code>expected 'defined id'</code> The identifier to be checked in an <code>#if</code> directive was not enclosed in parentheses.
C2004	<code>expected 'defined(id)'</code> An <code>#if</code> directive caused a syntax error.
C2005	<code>#line expected a line number</code> A <code>#line</code> directive lacked the required line-number specification.
C2006	<code>#include expected a file name</code> An <code>#include</code> directive lacked the required file-name specification.
C2007	<code>#define syntax</code> A <code>#define</code> directive caused a syntax error.
C2008	<code>'character' : unexpected in macro definition</code> The given character was used incorrectly in a macro definition.

Number	Compilation-Error Message
C2009	reuse of macro formal ' <i>identifier</i> ' The given identifier was used twice in the formal-parameter list of a macro definition.
C2010	' <i>character</i> ' : unexpected in formal list The given character was used incorrectly in the formal-parameter list of a macro definition.
C2011	' <i>identifier</i> ' : definition too big The given macro definitions exceeded 256 bytes.
C2012	missing name following '<' An #include directive lacked the required file-name specification.
C2013	missing '>' The closing angle bracket (>) was missing from an #include directive.
C2014	preprocessor command must start as first non-whitespace Non-white-space characters appear before the number sign (#) of a preprocessor directive on the same line.
C2015	too many chars in constant A character constant containing more than one character or escape sequence was used.
C2016	no closing single quote A character constant was not enclosed in single quotation marks.
C2017	illegal escape sequence The character or characters after the escape character (\) did not form a valid escape sequence.
C2018	unknown character 'Ox <i>character</i> ' The given hexadecimal number does not correspond to a character.

Number	Compilation-Error Message
C2019	<p>expected preprocessor command, found '<i>character</i>'</p> <p>The given character followed a number sign (#), but it was not the first letter of a preprocessor directive.</p>
C2020	<p>bad octal number '<i>character</i>'</p> <p>The given character was not a valid octal digit.</p>
C2021	<p>expected exponent value, not '<i>character</i>'</p> <p>The given character was used as the exponent of a floating-point constant but was not a valid number.</p>
C2022	<p>'<i>number</i>' : too big for char</p> <p>The <i>number</i> was too large to be represented as a character.</p>
C2023	<p>divide by 0</p> <p>The second operand in a division operation (/) evaluated to zero, giving undefined results.</p>
C2024	<p>mod by 0</p> <p>The second operand in a remainder operation (%) evaluated to zero, giving undefined results.</p>
C2025	<p>'<i>identifier</i>' : enum/struct/union type redefinition</p> <p>The given identifier had already been used for an enumeration, structure, or union tag.</p>
C2026	<p>'<i>identifier</i>' : member of enum redefinition</p> <p>The given identifier had already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.</p>
C2028	<p>struct/union member needs to be inside a struct/union</p> <p>Structure and union members must be declared within the structure or union.</p> <p>This error may be caused by an enumeration declaration that contains a declaration of a structure member, as in the following example:</p>

Number	Compilation-Error Message
	<pre>enum a { january, february, int march; /* structure declaration: ** illegal */ };</pre>
C2029	<p><i>'identifier'</i> : bit-fields allowed only in structs</p> <p>Only structure types may contain bit fields.</p>
C2030	<p><i>'identifier'</i> : struct/union member redefinition</p> <p>The <i>identifier</i> was used for more than one member of the same structure or union.</p>
C2031	<p><i>'identifier'</i> : function cannot be struct/union member</p> <p>The given function was declared to be a member of a structure.</p> <p>To correct this error, use a pointer to the function instead.</p>
C2032	<p><i>'identifier'</i> : base type with near/far/huge not allowed</p> <p>The given structure or union member was declared with the near, far, or huge keyword.</p>
C2033	<p><i>'identifier'</i> : bit-field cannot have indirection</p> <p>The given bit field was declared as a pointer (*), which is not allowed.</p>
C2034	<p><i>'identifier'</i> : bit-field type too small for number of bits</p> <p>The number of bits specified in the bit-field declaration exceeded the number of bits in the given base type.</p>
C2035	<p>enum/struct/union <i>'identifier'</i> : unknown size</p> <p>The given structure or union had an undefined size.</p>
C2036	<p>left of <i>'member'</i> must have struct/union type</p> <p>The expression before the member-selection operator (<i>-></i>) was not a pointer to a structure or union type, or the</p>

Number	Compilation-Error Message
	expression before the member-selection operator (.) did not evaluate to a structure or union. In this message, <i>member</i> is a member designator in one of the following forms: -> <i>identifier</i> . <i>identifier</i>
C2037	left of '->' or '.' specifies undefined struct/union The expression before the member-selection operator (-> or .) identified a structure or union type that was not defined.
C2038	' <i>identifier</i> ' : not struct/union member The given identifier was used in a context that required a structure or union member.
C2039	'->' requires struct/union pointer The expression before the member-selection operator (->) was a structure or union name, not a pointer to a structure or union as expected.
C2040	'.' requires struct/union name The expression before the member-selection operator (.) was a pointer to a structure or union, not a structure or union name as expected.
C2041	keyword 'enum' illegal The enum keyword appeared in a structure or union declaration, or an enum type definition was not formed correctly.
C2042	signed/unsigned keywords mutually exclusive The signed and unsigned keywords may not appear in the same declaration.
C2043	illegal break A break statement is legal only when it appears within a do , for , while , or switch statement.
C2044	illegal continue A continue statement is legal only when it appears within a do , for , or while statement.

Number	Compilation-Error Message
C2045	<i>'identifier'</i> : label redefined The given label appeared before more than one statement in the same function.
C2046	illegal case The case keyword may appear only within a switch statement.
C2047	illegal default The default keyword may appear only within a switch statement.
C2048	more than one default A switch statement contained more than one default label.
C2049	cast has illegal formal parameter list A formal parameter list was given in a type-cast expression.
C2050	non-integral switch expression A switch expression was not integral.
C2051	case expression not constant Case expressions must be integral constants.
C2052	case expression not integral Case expressions must be integral constants.
C2053	case value <i>number</i> already used The given case value was already used in this switch statement.
C2054	expected '(' to follow <i>'identifier'</i> The context requires parentheses after the function <i>identifier</i> .
C2055	expected formal parameter list, not a type list An argument-type list appeared in a function definition instead of a formal parameter list.

Number	Compilation-Error Message
C2056	illegal expression An expression was illegal because of a previous error. (The previous error may not have produced an error message.)
C2057	expected constant expression The context requires a constant expression.
C2058	constant expression is not integral The context requires an integral constant expression.
C2059	syntax error : ' <i>token</i> ' The given token caused a syntax error.
C2060	syntax error : EOF The end of the file was encountered unexpectedly, causing a syntax error. This error can be caused by a missing closing curly brace (}) at the end of your program.
C2061	syntax error : identifier ' <i>identifier</i> ' The given identifier caused a syntax error.
C2062	type ' <i>type</i> ' unexpected The given type was misused.
C2063	' <i>identifier</i> ' : not a function The given identifier was not declared as a function, but an attempt was made to use it as a function.
C2064	term does not evaluate to a function An attempt was made to call a function through an expression that did not evaluate to a function pointer.
C2065	' <i>identifier</i> ' : undefined The given identifier was not defined.
C2066	cast to function returning . . . is illegal An object was cast to a function type.
C2067	cast to array type is illegal An object was cast to an array type.

Number	Compilation-Error Message
C2068	<p>illegal cast</p> <p>A type used in a cast operation was not a legal type.</p>
C2069	<p>cast of 'void' term to non-void</p> <p>The void type was cast to a different type.</p>
C2070	<p>illegal sizeof operand</p> <p>The operand of a sizeof expression was not an identifier or a type name.</p>
C2071	<p>'class' : bad storage class</p> <p>The given storage class cannot be used in this context.</p>
C2072	<p>'identifier' : initialization of a function</p> <p>An attempt was made to initialize a function.</p>
C2073	<p>'identifier' : cannot initialize array in function</p> <p>An attempt was made to initialize the given array within a function. Arrays can be initialized only at the external level.</p>
C2074	<p>cannot initialize struct/union in function</p> <p>An attempt was made to initialize the given structure or union within a function. Structures and unions can be initialized only at the external level.</p>
C2075	<p>'identifier' : array initialization needs curly braces</p> <p>The braces ({ }) around the given array initializer were missing.</p>
C2076	<p>'identifier' : struct/union initialization needs curly braces</p> <p>The braces ({ }) around the given structure or union initializer were missing.</p>
C2077	<p>non-integral field initializer 'identifier'</p> <p>An attempt was made to initialize a bit-field member of a structure with a nonintegral value.</p>
C2078	<p>too many initializers</p> <p>The number of initializers exceeded the number of objects to be initialized.</p>

Number	Compilation-Error Message
C2079	<p><i>'expression'</i> uses undefined struct/union</p> <p>The given identifier was declared as a structure or union type that had not been defined.</p>
C2082	<p>redefinition of formal parameter <i>'identifier'</i></p> <p>A formal parameter to a function was redeclared within the function body.</p>
C2083	<p>array <i>'identifier'</i> already has a size</p> <p>The dimensions of the given array had already been declared.</p>
C2084	<p>function <i>'identifier'</i> already has a body</p> <p>The given function had already been defined.</p>
C2085	<p><i>'identifier'</i> : not in formal parameter list</p> <p>The given parameter was declared in a function definition for a nonexistent formal parameter.</p>
C2086	<p><i>'identifier'</i> : redefinition</p> <p>The given identifier was defined more than once.</p>
C2087	<p><i>'identifier'</i> : missing subscript</p> <p>The definition of an array with multiple subscripts was missing a subscript value for a dimension other than the first dimension, as in the following example:</p> <pre>int func(a) char a[10] []; /* Illegal */ { . . . } int func(a) char a[] [5]; /* Legal */ { . . . }</pre>

Number	Compilation-Error Message
C2088	<p>use of undefined enum/struct/union '<i>identifier</i>'</p> <p>The given identifier referred to a structure or union type that was not defined.</p>
C2089	<p>typedef specifies a near/far function</p> <p>The use of the near or far keyword in a typedef declaration conflicted with the use of near or far for the declared item, as in the following example:</p> <pre>typedef int far FARFUNC(); FARFUNC near *fp;</pre>
C2090	<p>function returns array</p> <p>A function cannot return an array. (It can return a pointer to an array.)</p>
C2091	<p>function returns function</p> <p>A function cannot return a function. (It can return a pointer to a function.)</p>
C2092	<p>array element type cannot be function</p> <p>Arrays of functions are not allowed; however, arrays of <i>pointers</i> to functions are allowed.</p>
C2093	<p>cannot initialize a static or struct with address of automatic vars</p> <p>You cannot use the address of an auto variable in the initializer of a static item.</p>
C2094	<p>label '<i>identifier</i>' was undefined</p> <p>The function did not contain a statement labeled with the given identifier.</p>
C2095	<p><i>function:</i> actual has type void: parameter number</p> <p>An attempt was made to pass a void argument to a function. Formal parameters and arguments to functions cannot have type void; they can, however, have type void * (pointer to void).</p>
C2096	<p>struct/union comparison illegal</p> <p>You cannot compare two structures or unions. (You can, however, compare individual members within structures and unions.)</p>

Number	Compilation-Error Message
C2097	illegal initialization An attempt was made to initialize a variable using a non-constant value.
C2098	non-address expression An attempt was made to initialize an item that was not an lvalue.
C2099	non-constant offset An initializer used a nonconstant offset.
C2100	illegal indirection The indirection operator (*) was applied to a nonpointer value.
C2101	'&' on constant The address-of operator (&) did not have an lvalue as its operand.
C2102	'&' requires lvalue The address-of operator must be applied to an lvalue expression.
C2103	'&' on register variable An attempt was made to take the address of a register variable.
C2104	'&' on bit-field An attempt was made to take the address of a bit field.
C2105	'operator' needs lvalue The given operator did not have an lvalue operand.
C2106	'operator' : left operand must be lvalue The left operand of the given operator was not an lvalue.
C2107	illegal index, indirection not allowed A subscript was applied to an expression that did not evaluate to a pointer.

Number	Compilation-Error Message
C2108	non-integral index A nonintegral expression was used in an array subscript.
C2109	subscript on non-array A subscript was used on a variable that was not an array.
C2110	'+' : 2 pointers An attempt was made to add one pointer to another.
C2111	pointer + non-integral value An attempt was made to add a nonintegral value to a pointer.
C2112	illegal pointer subtraction An attempt was made to subtract pointers that did not point to the same type.
C2113	'-' : right operand pointer The right operand in a subtraction operation (—) was a pointer, but the left operand was not.
C2114	'operator' : pointer on left; needs integral right The left operand of the given operator was a pointer; the right operand must be an integral value.
C2115	'identifier' : incompatible types An expression contained incompatible types.
C2116	operator : bad left (or right) operand The specified operand of the given operator was illegal for that operator.
C2117	'operator' : illegal for struct/union Structure and union type values are not allowed with the given operator.
C2118	negative subscript A value defining an array size was negative.

Number	Compilation-Error Message
C2119	<p>'typedefs' both define indirection</p> <p>Two typedef types were used to declare an item and both typedef types had indirection. For example, the declaration of p in the following example is illegal:</p> <pre>typedef int *P_INT; typedef short *P_SHORT; /* this declaration is illegal */ P_SHORT P_INT p;</pre>
C2120	<p>'void' illegal with all types</p> <p>The void type was used in a declaration with another type.</p>
C2121	<p>typedef specifies different enum</p> <p>An attempt was made to use a type declared in a typedef statement to specify both an enumeration type and another type.</p>
C2122	<p>typedef specifies different struct</p> <p>An attempt was made to use a type declared in a typedef statement to specify both a structure type and another type.</p>
C2123	<p>typedef specifies different union</p> <p>An attempt was made to use a type declared in a typedef statement to specify both a union type and another type.</p>
C2125	<p><i>identifier</i> : allocation exceeds 64K</p> <p>The given item exceeds the size limit of 64K.</p> <p>The only items that are allowed to exceed 64K are huge arrays.</p>
C2126	<p><i>identifier</i> : automatic allocation exceeds 32K</p> <p>The space allocated for the local variables of a function exceeded the limit of 32K.</p>
C2127	<p>parameter allocation exceeds 32K</p> <p>The storage space required for the parameters to a function exceeded the limit of 32K.</p>

Number	Compilation-Error Message
C2128	<i>identifier</i> : huge array cannot be aligned to segment boundary The given array violated one of the restrictions imposed on huge arrays; see Section 6.3.5, "Creating Huge-Model Programs," for more information on these restrictions.
C2129	static function ' <i>identifier</i> ' not found A forward reference was made to a static function that was never defined.
C2130	#line expected a string containing the file name A file name was missing from a #line directive.
C2131	attributes specify more than one near/far/huge More than one near , far , or huge attribute was applied to an item, as in the following example: <pre>typedef int near NINT; NINT far a; /* Illegal */</pre>
C2132	syntax error : unexpected identifier An identifier appeared in a syntactically illegal context.
C2133	array ' <i>identifier</i> ' : unknown size An attempt was made to declare an unsized array as local variable, as in the following example: <pre>int mat_add(array1) int array1[]; /* Legal */ { int array2[]; /* Illegal */ . . . }</pre>
C2134	<i>identifier</i> : struct/union too large The size of a structure or union exceeded the compiler limit (2 ³² bytes).

Number	Compilation-Error Message
C2135	<p>missing ')' in macro expansion</p> <p>A macro reference with arguments was missing a closing parenthesis ()).</p>
C2137	<p>empty character constant</p> <p>The illegal character constant '' was used.</p>
C2138	<p>unmatched close comment '/*'</p> <p>The compiler detected an open-comment delimiter (/*) without a matching close-comment delimiter (*//).</p> <p>This error usually indicates an attempt to use illegal nested comments.</p>
C2139	<p>type following 'type' is illegal</p> <p>An illegal type combination such as the following was used:</p> <p>long char a;</p>
C2140	<p>argument type cannot be function returning ...</p> <p>A function was declared as a formal parameter of another function, as in the following example:</p> <pre>int func1(a) int a(); /* Illegal */</pre>
C2141	<p>value out of range for enum constant</p> <p>An enumeration constant had a value outside the range of values allowed for type <code>int</code>.</p>
C2142	<p>ellipsis requires three periods</p> <p>The compiler detected the token <code>".."</code> and assumed that <code>"..."</code> was intended.</p>
C2143	<p>syntax error : missing '<i>token1</i>' before '<i>token2</i>'</p> <p>The compiler expected <i>token1</i> to appear before <i>token2</i>. This message may appear if a required closing curly brace <code>}</code>, right parenthesis <code>)</code>, or semicolon <code>;</code> is missing.</p>

Number	Compilation-Error Message
C2144	<p>syntax error : missing '<i>token</i>' before type '<i>type</i>'</p> <p>The compiler expected the given token to appear before the given type name. This message may appear if a required closing curly brace (}), right parenthesis ()), or semicolon (;) is missing.</p>
C2145	<p>syntax error : missing '<i>token</i>' before identifier</p> <p>The compiler expected the given token to appear before an identifier. This message may appear if a semicolon (;) does not appear after the last declaration of a block.</p>
C2146	<p>syntax error : missing '<i>token</i>' before identifier '<i>identifier</i>'</p> <p>The compiler expected the given token to appear before the given identifier.</p>
C2147	<p>array : unknown size</p> <p>An attempt was made to increment an index or pointer to an array whose base type has not yet been declared.</p>
C2148	<p>array too large</p> <p>An array exceeded the maximum legal size (2^{32} bytes).</p>
C2149	<p><i>identifier</i> : named bit-field cannot have 0 width</p> <p>The given named bit field had a zero width. Only unnamed bit fields are allowed to have zero width.</p>
C2150	<p><i>identifier</i> : bit-field must have type int, signed int, or unsigned int</p> <p>The ANSI C standard requires bit fields to have types of int, signed int, or unsigned int. This message appears only if you compiled your program with the /Za option.</p>
C2151	<p>more than one cdecl/fortran/pascal attribute specified</p> <p>More than one keyword specifying a function-calling convention was given.</p>

Number	Compilation-Error Message
C2152	<i>identifier</i> : pointers to functions with different attributes An attempt was made to assign a pointer to a function declared with one calling convention (cdecl , fortran , or pascal) to a pointer to a function declared with a different calling convention.
C2153	hex constants must have at least 1 hex digit At least one hexadecimal digit must follow the "x". The hexadecimal constants 0x and 0X are illegal.
C2154	' <i>name</i> ' : does not refer to a segment The <i>name</i> was the first identifier given in an alloc_text pragma argument list and it is already defined as something other than a segment name.
C2155	' <i>name</i> ' : already in a segment The function <i>name</i> appears in more than one alloc_text pragma.
C2156	pragma must be at outer level Certain pragmas must be specified at a global level, outside a function body, and there is an occurrence of one of these pragmas within a function.
C2157	' <i>name</i> ' : must be declared before use in pragma list The function <i>name</i> in the list of functions for an alloc_text pragma has not been declared prior to being referenced in the list.
C2158	' <i>name</i> ' : is a function <i>Name</i> was specified in the list of variables in a same_segment pragma, but was previously declared as a function.
C2159	more than one storage class specified Illegal declaration—only one storage class is allowed.

Number	Compilation-Error Message
C2160	<p>## cannot occur at the beginning of a macro definition</p> <p>A macro definition cannot begin with a token-pasting (<code>##</code>) operator.</p>
C2161	<p>## cannot occur at the end of a macro definition</p> <p>A macro definition cannot end with a token-pasting (<code>##</code>) operator.</p>
2162	<p>expected macro formal parameter</p> <p>The token following a stringizing operator (<code>#</code>) must be a formal parameter name.</p>
2163	<p>'<i>string</i>' : not available as an intrinsic</p> <p>A function specified in the list of functions for an intrinsic or function pragma is not one of the functions available in intrinsic form.</p>
C2165	<p>'<i>keyword</i>' : cannot modify pointers to data</p> <p>Bad use of fortran, pascal or cdecl keyword to modify pointer to data.</p>
C2167	<p>'<i>name</i>' : too many actual parameters for intrinsic</p> <p>A reference to the intrinsic function <i>name</i> contains too many actual parameters.</p>
C2168	<p>'<i>name</i>' : too few actual parameters for intrinsic</p> <p>A reference to the intrinsic function <i>name</i> contains too few actual parameters.</p>
C2169	<p>'<i>name</i>' : is an intrinsic, it cannot be defined</p> <p>An attempt was made to provide a function definition for a function already declared as an intrinsic.</p>
C2170	<p><i>identifier</i> : intrinsic not declared as a function</p> <p>You tried to use the intrinsic pragma for an item other than a function, or for a function that does not have an intrinsic form. (The section titled "Generating Intrinsic</p>

Number	Compilation-Error Message
	Functions" in Section 3.3.15 lists the functions that have intrinsic forms.)
C2177	constant too big Information was lost because a constant value was too large to be represented in the type to which it was assigned. (1)
C2171	'operator' : bad operand Illegal operand type for the specified unary operator.

E.3.3 Warning Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in parentheses at the end of each warning-message description gives the minimum warning level that must be set for the message to appear.

Number	Warning Message
C4000	UNKNOWN WARNING Contact Microsoft Technical Support. The compiler detected an unknown error condition. Please report this condition to Microsoft Corporation, using the Product Assistance Request form at the back of this manual.
C4001	macro ' <i>identifier</i> ' requires parameters The given identifier was defined as a macro taking one or more arguments, but it was used in the program without arguments. (1)
C4002	too many actual parameters for macro ' <i>identifier</i> ' The number of actual arguments specified with the given identifier was greater than the number of formal parameters given in the macro definition of the identifier. (1)
C4003	not enough actual parameters for macro ' <i>identifier</i> ' The number of actual arguments specified with the given identifier was less than the number of formal parameters given in the macro definition of the identifier. (1)

Number	Warning Message
C4004	missing close parenthesis after 'defined' The closing parenthesis was missing from an #if defined phrase. (1)
C4005	' <i>identifier</i> ' : redefinition The given identifier was redefined. (1)
C4006	#undef expected an identifier The name of the identifier whose definition was to be removed was not given with the #undef directive. (1)
C4009	string too big, trailing chars truncated A string exceeded the compiler limit on string size. To correct this problem, break the string into two or more strings. (1)
C4011	identifier truncated to ' <i>identifier</i> ' Only the identifier's first 31 characters are significant. (1)
C4014	' <i>identifier</i> ' : bit-field type must be unsigned The given bit field was not declared as an unsigned type. Bit fields must be declared as unsigned integral types. A conversion has been supplied. (1)
C4015	' <i>identifier</i> ' : bit-field type must be integral The given bit field was not declared as an integral type. Bit fields must be declared as unsigned integral types. A conversion has been supplied. (1)
C4016	' <i>identifier</i> ' : no function return type The given function had not yet been declared or defined, so the return type was unknown. The default return type (int) is assumed. (2)

Number	Warning Message
C4017	<p>cast of int expression to far pointer</p> <p>A far pointer represents a full segmented address. On an 8086/8088 processor, casting an int value to a far pointer may produce an address with a meaningless segment value. (1)</p>
C4020	<p>too many actual parameters</p> <p>The number of arguments specified in a function call was greater than the number of parameters specified in the argument-type list or function definition. (1)</p>
C4021	<p>too few actual parameters</p> <p>The number of arguments specified in a function call was less than the number of parameters specified in the argument-type list or function definition. (1)</p>
C4022	<p>pointer mismatch : parameter <i>n</i></p> <p>The pointer type of the given parameter was different from the pointer type specified in the argument-type list or function definition. (1)</p>
C4024	<p>different types : parameter <i>n</i></p> <p>The type of the given parameter in a function call did not agree with the type given in the argument-type list or function definition. (1)</p>
C4025	<p>function declaration specified variable argument list</p> <p>The argument-type list in a function declaration ended with a comma or a comma followed by ellipsis dots (...), indicating that the function could take a variable number of arguments, but no formal parameters were declared for the function. (1)</p>
C4026	<p>function was declared with formal argument list</p> <p>The function was declared to take arguments, but the function definition did not declare formal parameters. (1)</p>

Number	Warning Message
C4027	<p>function was declared without formal argument list</p> <p>The function was declared to take no arguments (the argument-type list consisted of the word void), but formal parameters were declared in the function definition or arguments were given in a call to the function. (1)</p>
C4028	<p>parameter <i>n</i> declaration different</p> <p>The type of the given parameter did not agree with the corresponding type in the argument-type list or with the corresponding formal parameter. (1)</p>
C4029	<p>declared parameter list different from definition</p> <p>The argument-type list given in a function declaration did not agree with the types of the formal parameters given in the function definition. (1)</p>
C4030	<p>first parameter list is longer than the second</p> <p>A function was declared more than once with different argument-type lists in the declarations. (1)</p>
C4031	<p>second parameter list is longer than the first</p> <p>A function was declared more than once with different argument-type lists. (1)</p>
C4032	<p>unnamed struct/union as parameter</p> <p>The structure or union type being passed as an argument was not named, so the declaration of the formal parameter cannot use the name and must declare the type. (1)</p>
C4033	<p>function must return a value</p> <p>A function is expected to return a value unless it is declared as void. (2)</p>
C4034	<p>sizeof returns 0</p> <p>The sizeof operator was applied to an operand that yielded a size of zero. (1)</p>

Number	Warning Message
C4035	<i>identifier</i> : no return value A function declared to return a value did not do so. (2)
C4036	unexpected formal parameter list A formal parameter list was given in a function declaration. The formal parameter list is ignored. (1)
C4037	' <i>identifier</i> ' : formal parameters ignored No storage class or type name appeared before the declarators of formal parameters in a function declaration, as in the following example: <pre>int *f(a,b,c);</pre> The formal parameters are ignored. (1)
C4038	' <i>identifier</i> ' : formal parameter has bad storage class The given formal parameter was declared with a storage class other than auto or register . (1)
C4039	' <i>identifier</i> ' : function used as an argument A formal parameter to a function was declared to be a function, which is illegal. The formal parameter is converted to a function pointer. (1)
C4040	near/far/huge on ' <i>identifier</i> ' ignored The near or far keyword has no effect in the declaration of the given identifier and is ignored. (1)
C4041	formal parameter ' <i>identifier</i> ' is redefined The given formal parameter was redefined in the function body, making the corresponding actual argument unavailable in the function. (1)
C4042	' <i>identifier</i> ' : has bad storage class The specified storage class cannot be used in this context (for example, function parameters cannot be given extern class). The default storage class for that context was used in place of the illegal class. (1)

Number	Warning Message
C4043	<p><i>'identifier'</i> : void type changed to int</p> <p>An item other than a function was declared to have void type. (1)</p>
C4044	<p>huge on <i>'identifier'</i> ignored, must be an array</p> <p>The huge keyword was used to declare the given nonarray item. (1)</p>
C4045	<p><i>'identifier'</i> : array bounds overflow</p> <p>Too many initializers were present for the given array. The excess initializers are ignored. (1)</p>
C4046	<p><i>'&'</i> on function/array, ignored</p> <p>An attempt was made to apply the address-of operator (&) to a function or array identifier. (1)</p>
C4047	<p><i>'operator'</i> : different levels of indirection</p> <p>An expression involving the specified operator had inconsistent levels of indirection. (1)</p> <p>The following example illustrates this condition:</p> <pre>char **p; char *q; . . . p = q;</pre>
C4048	<p>array's declared subscripts different</p> <p>An array was declared twice with different sizes. The larger size is used. (1)</p>
C4049	<p><i>'operator'</i> : indirection to different types</p> <p>The indirection operator (*) was used in an expression to access values of different types. (1)</p>

Number	Warning Message
C4051	<p>data conversion</p> <p>Two data items in an expression had different types, causing the type of one item to be converted. (2)</p>
C4052	<p>different enum types</p> <p>Two different enum types were used in an expression. (1)</p>
C4053	<p>at least one void operand</p> <p>An expression with type void was used as an operand. (1)</p>
C4056	<p>overflow in constant arithmetic</p> <p>The result of an operation exceeded 0x7FFFFFFF. (1)</p>
C4057	<p>overflow in constant multiplication</p> <p>The result of an operation exceeded 0x7FFFFFFF. (1)</p>
C4058	<p>address of frame variable taken, DS != SS</p> <p>The program was compiled with the default data segment (DS) not equal to the stack segment (SS), and the program tried to point to a frame variable with a near pointer. (1)</p>
C4059	<p>segment lost in conversion</p> <p>The conversion of a far pointer (a full segmented address) to a near pointer (a segment offset) resulted in the loss of the segment address. (1)</p>
C4060	<p>conversion of long address to short address</p> <p>The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) resulted in the loss of the segment address. (1)</p>
C4061	<p>long/short mismatch in argument: conversion supplied</p> <p>The base types of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)</p>

Number	Warning Message
C4062	<p>near/far mismatch in argument: conversion supplied</p> <p>The pointer sizes of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)</p>
C4063	<p>'<i>identifier</i>' : function too large for post-optimizer</p> <p>The given function was not optimized because not enough space was available. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)</p>
C4064	<p>procedure too large, skipping <i>description</i> optimization and continuing</p> <p>Some optimizations for a function were skipped because not enough space was available for optimization. (0)</p> <p>To correct this problem, reduce the size of the function by dividing it into two or more smaller functions.</p> <p>The <i>description</i> in this message may appear as any of the following:</p> <p>loop inversion branch sequence cross jump</p>
C4065	<p>recoverable heap overflow in post-optimizer - some optimizations may be missed</p> <p>Some optimizations were skipped because not enough space was available for optimization. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)</p>
C4066	<p>local symbol table overflow - some local symbols may be missing in listings</p> <p>The listing generator ran out of heap space for local variables, so the source listing may not contain symbol-table information for all local variables.</p>

Number	Warning Message
C4067	<p>unexpected characters following '<i>directive</i>' <i>directive</i> - newline expected</p> <p>Extra characters followed a preprocessor directive, as in the following example (1):</p> <pre>#endif NO_EXT_KEYS</pre> <p>This is accepted in Version 3.0, but not in Versions 4.0 and 5.0. Versions 4.0 and 5.0 require comment delimiters, such as the following:</p> <pre>#endif /* NO_EXT_KEYS */</pre>
C4068	<p>unknown pragma</p> <p>The compiler did not recognize a pragma and ignored it. (1)</p>
C4069	<p>conversion of near pointer to long integer</p> <p>A near pointer was converted to a long integer, which involves first extending the high-order word with the current data-segment value, <i>not</i> 0 as in Version 3.0. (1)</p>
C4071	<p>'<i>identifier</i>' : no function prototype given</p> <p>The given function was called before the compiler found the corresponding function prototype. (3)</p>
C4072	<p>Insufficient memory to process debugging information</p> <p>You compiled the program with the /Zi option, but not enough memory was available to create the required debugging information. (1)</p>
C4073	<p>scoping too deep, deepest scoping merged when debugging</p> <p>Declarations appeared at a static nesting level greater than 13. As a result, all declarations will seem to appear at the same level. (1)</p>
C4074	<p>non standard extension used - '<i>extension</i>'</p> <p>The given nonstandard language extension was used when the /Ze option was in effect. These extensions are given in Section 3.3.14, "Enabling and Disabling Language Extensions." (If the /Za option is in effect, this condition generates an error.) (3)</p>

Number	Warning Message
C4075	<p>size of switch expression or case constant too large - converted to int</p> <p>A value appearing in a switch or case statement was larger than an int. The compiler converts the illegal value to an int. (1)</p>
C4076	<p>'<i>type</i>' : may be used on integral types only</p> <p>The type modifiers signed and unsigned can be combined only with other integral types.</p>
C4077	<p>unknown check_stack option</p> <p>Unknown option given when using the old form of the check_stack pragma. The option must be empty, +, or -.</p>
C4079	<p>unexpected char '<i>character</i>'</p> <p>Unexpected separator <i>character</i> found in argument list of a pragma.</p>
C4080	<p>missing segment name</p> <p>The first argument in the argument list for the alloc_text pragma is missing a segment name. This happens if the first token in the argument list is not an identifier.</p>
C4081	<p>expected a comma</p> <p>There is a missing comma (,) between two arguments of a pragma.</p>
C4082	<p>expected an identifier</p> <p>There is a missing identifier in list of arguments to a pragma.</p>
C4083	<p>missing '('</p> <p>There is a missing opening parenthesis (() in the argument list for a pragma.</p>
C4084	<p>expected a pragma keyword</p> <p>The token following the pragma keyword is not an identifier.</p>
C4085	<p>expected [on off]</p> <p>Bad argument given for new form of check_stack pragma.</p>

Number	Warning Message
C4086	expected [1 2 4] Bad argument given for pack pragma.
C4087	' <i>name</i> ' : declared with <i>void</i> parameter list The function <i>name</i> was declared as taking no parameters, but a call to the function specifies actual parameters.
C4090	different ' <i>const</i> ' attributes The program passed a pointer to a <i>const</i> item to a function where the corresponding formal parameter is a pointer to a non- <i>const</i> item, which means the item could be modified by the function undetected.
C4091	no symbols were declared An empty declaration was detected. (2)
C4092	untagged enum/struct/union declared no symbols An empty declaration was detected that used an untagged enum/struct/union. (2)
C4093	unescaped newline in character constant in non-active code The constant expression of an #if , #elif , #ifdef , or #ifndef preprocessor directive evaluated to 0, making the following code inactive, and a new-line character appeared between a single or double quotation mark and the matching single or double quotation mark in that inactive code.
C4094	unexpected newline A new-line character appeared in a pragma where a comma, right parenthesis, or identifier was expected, as in the following examples: <pre>#pragma intrinsic (memset #pragma intrinsic (memset,</pre>
C4095	too many arguments for pragma More than one argument was given for a pragma that can take only one argument.

E.3.4 Compiler Limits

To operate the Microsoft C Optimizing Compiler, you must have sufficient disk space available for the compiler to create temporary files used in processing. The space required is approximately two times the size of the source file.

Table E.1 summarizes the limits imposed by the C compiler. If your program exceeds one of these limits, an error message will inform you of the problem.

Table E.1

Limits Imposed by the C Compiler

Program Item	Description	Limit
String literals	Maximum length of a string, including the terminating null character (<code>\0</code>)	512 bytes
Constants	Maximum size of a constant is determined by its type; see the <i>Microsoft C Language Reference</i> for a discussion of constants.	
Identifiers	Maximum length of an identifier	31 bytes (additional characters are discarded)
Declarations	Maximum level of nesting for structure/union definitions	10 levels
Preprocessor directives	Maximum size of a macro definition	512 bytes
	Maximum number of actual arguments to a macro definition	8 arguments
	Maximum length of an actual preprocessor argument	256 bytes
	Maximum level of nesting for <code>#if</code> , <code>#ifdef</code> , and <code>#ifndef</code> directives	32 levels
	Maximum level of nesting for include files	10 levels

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

E.4 Run-Time Error Messages

Run-time error messages fall into the following four categories:

1. Error messages generated by the run-time library to notify you of serious errors. These messages are listed and described in Section E.4.1.
2. Floating-point exceptions generated by the 8087/80287 hardware or the emulator. These exceptions are listed and described in Section E.4.2.
3. Error messages generated by program calls to error-handling routines in the C run-time library (the **abort**, **assert**, and **perror** routines). These routines print an error message to standard error whenever the program calls the given routine. For descriptions of these routines and the corresponding error messages, see the *Microsoft C Run-Time Library Reference*.
4. Error messages generated by calls to math routines in the C run-time library. On error, the math routines return an error value and some print a message to the standard error. See the *Microsoft C Run-Time Library Reference* for descriptions of the math routines and corresponding error messages.

E.4.1 Run-Time-Library Error Messages

The following messages may be generated at run time when your program has serious errors. Run-time error-message numbers range from R6000 to R6999.

A run-time error message takes the following general form:

```
run-time error R6nnn
- messagetext
```

Number	Run-Time-Library Error Message
--------	--------------------------------

R6000	stack overflow
-------	----------------

Your program has run out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive. The program was terminated with an exit code of 255.

Number Run-Time-Library Error Message

To correct the problem, recompile using the **/F** option of the **CL** command or relink using the linker **/STACK** option to allocate a large stack, or modify the stack information in the executable-file header by using the **EXE-MOD** program. (See Chapter 15 of the Microsoft CodeView and Utilities manual for information about the **EXEMOD** program.)

R6001 null pointer assignment

The contents of the **NULL** segment have changed in the course of program execution. The **NULL** segment is a special location in low memory that is not normally used. If the contents of the **NULL** segment change during a program's execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. Note that your program can contain null pointers without generating this message; the message appears only when you access a memory location through the null pointer.

This error does not cause your program to terminate; the error message is printed following the normal termination of the program. This error yields a nonzero exit code.

This message reflects a potentially serious error in your program. Although a program that produces this error may appear to operate correctly, it is likely to cause problems in the future and may fail to run in a different operating environment.

R6002 floating point not loaded

Your program needs the floating-point library, but the library was not loaded. The error causes the program to be terminated with an exit status of 255. This occurs in two situations:

1. The program was compiled or linked with an option (such as **/FPI87**) that required an 8087 or 80287 coprocessor, but the program was run on a machine that did not have a coprocessor installed.

To fix this problem, either recompile the program with the **/FPI** option, relink with an emulator library (**mLIBCE.LIB**), or install a coprocessor. (See Section 3.3.1 of this manual for more information about these options and libraries.)

Number	Run-Time-Library Error Message
	<p>2. A format string for one of the routines in the printf or scanf families contains a floating-point format specification and there are no floating-point values or variables in the program. The C compiler attempts to minimize the size of a program by loading floating-point support only when necessary. Floating-point format specifications within format strings are not detected, so the necessary floating-point routines are not loaded.</p> <p>To correct this error, use a floating-point argument to correspond to the floating-point format specification. This causes floating-point support to be loaded.</p>
R6003	<p>integer divide by 0</p> <p>An attempt was made to divide an integer by 0, giving an undefined result. This error terminates the program with an exit code of 255.</p>
R6004	<p>DOS 2.0 or later required</p> <p>The C compiler cannot run on versions of MS-DOS prior to Version 2.0.</p>
R6005	<p>not enough memory on exec</p> <p>Errors R6005 through R6007 occur when a child process spawned by one of the exec library routines fails and MS-DOS could not return control to the parent process. This error indicates that not enough memory remained to load the program being spawned.</p>
R6006	<p>bad format on exec</p> <p>The file to be executed by one of the exec functions was not in the correct format for an executable file.</p>
R6007	<p>bad environment on exec</p> <p>During a call to one of the exec functions, MS-DOS determined that the child process was being given a bad environment block.</p>
R6008	<p>not enough space for arguments</p> <p>See explanation under error R6009.</p>

Number Run-Time-Library Error Message

R6009 not enough space for environment

Errors R6008 and R6009 both occur at start-up if there is enough memory to load the program, but not enough room for the **argv** vector, the **envp** vector, or both. To avoid this problem, rewrite the **_setargv** or **_setenvp** routines (see Section 5.2.2, "Suppressing Command-Line Processing," for more information).

E.4.2 Floating-Point Exceptions

The error messages listed below correspond to exceptions generated by the 8087/80287 hardware. Refer to the Intel documentation for your processor for a detailed discussion of hardware exceptions. These errors may also be detected by the floating-point emulator or alternate math library.

If you use the C-language default 8087/80287 control-word settings, the following exceptions are masked and do not occur:

Exception	Default Masked Action
Denormal	Exception masked.
Underflow	Result goes to 0.0.
Inexact	Exception masked.

For information on how to change the floating-point control word, see the reference pages for **_control87** in the *Microsoft C Run-Time Library Reference*.

The following errors do not occur with code generated by the Microsoft C Optimizing Compiler or provided in the Microsoft C Run-Time Library:

Square root
Stack underflow
Unemulated

The floating-point exceptions have the following format:

```
run-time error M61nn: MATH
- floating-point error:  messagetext
```


The floating-point exceptions are listed and described below.

Number	Floating-Point Exception
M6101	<code>invalid</code> An invalid operation occurred. This usually involves operating on a NAN or an infinity. This error terminates the program with exit code 129.
M6102	<code>denormal</code> A very small floating-point number was generated, which may no longer be valid due to loss of significance. Denormals are normally masked, causing them to be trapped and operated on. This error terminates the program with exit code 130.
M6103	<code>divide by 0</code> An attempt was made to divide by zero. This error terminates the program with exit code 131.
M6104	<code>overflow</code> An overflow occurred in a floating-point operation. This error terminates the program with exit code 132.
M6105	<code>underflow</code> An underflow occurred in a floating-point operation. (An underflow is normally masked so that the underflowing value is replaced with 0.0.) This error terminates the program with exit code 133.
M6106	<code>inexact</code> Loss of precision occurred in a floating-point operation. This exception is normally masked, since almost any floating-point operation can cause loss of precision. This error terminates the program with exit code 134.
M6107	<code>unemulated</code> An attempt was made to execute an 8087/80287 instruction that is invalid or is not supported by the emulator. This error terminates the program with exit code 135.
M6108	<code>square root</code> The operand in a square-root operation was negative. This error terminates the program with exit code 136. (Note: the sqrt function in the C run-time library checks the argument before performing the operation and returns an error

Number Floating-Point Exception

value if the operand is negative; see the *Microsoft C Run-Time Library Reference* for details on **sqrt**.)

- M6110 **stack overflow**
 A floating-point expression caused a stack overflow on the 8087 or 80287 coprocessor or the emulator. (Stack-overflow exceptions are trapped up to a limit of seven levels in addition to the eight levels normally supported by the 8087 or 80287 coprocessor.) This error terminates the program with exit code 138.
- M6111 **stack underflow**
 A floating-point operation resulted in a stack underflow on the 8087 or 80287 coprocessor or the emulator. This error terminates the program with exit code 139.
- M6112 **explicitly generated**
 A signal indicating a floating-point error was sent using a **raise (SIGFPE)** call. This error terminates the program with exit code 140.

E.4.3 Run-Time Limits

Table E.2 summarizes the limits that apply to programs at run time. If your program exceeds one of these limits, an error message will inform you of the problem.

Table E.2
Program Limits at Run Time

Item	Description	Limit
Files	Maximum file size	$2^{32} - 1$ bytes (4 gigabytes)
	Maximum number of open files (streams)	20^a
Command line	Maximum number of characters (including program name)	128
Environment table	Maximum size	32K

^a Five streams are opened automatically (**stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**), leaving 15 files available for the program to open.

The definitions in this glossary are intended primarily for use with this manual, the *Microsoft C Language Reference*, and the *Microsoft C Run-Time Library Reference*. Neither individual definitions nor the list of terms is comprehensive.

8087 or 80287 coprocessor

Intel® hardware products that provide very fast and precise number processing.

abstract declarator

A declarator without an identifier, consisting of a type and, optionally, one or more pointer, array, or function modifiers.

aggregate types

Arrays, structures, and unions.

alias

One of several alternative names for the same memory location.

alternate math library

A model-dependent floating-point library that uses a subset of the Institute of Electrical and Electronics Engineers, Inc. (IEEE) number format. Linking with this library results in the smallest, fastest programs available without a coprocessor, but sacrifices some accuracy in results for speed.

ANSI (American National Standards Institute)

The national institute responsible for defining programming-language standards to promote portability of these languages between different computer systems.

argument

A value passed to a function.

argument-type list

In a function prototype, a list of abstract declarators, separated by commas, indicating the types of actual arguments in the function call. Used to make sure the actual arguments in the function call correspond to the formal parameters in the function definition.

argc

The traditional name for the first argument to the **main** function in a C source program: an integer specifying how many arguments are passed to the program from the command line.

argv

The traditional name for the second argument to the **main** function in a C source program: a pointer to an array of strings. Traditionally, the first string is the program name and each following string is an argument passed to the program from the command line.

arithmetic conversion

Conversion operations performed on items of integral and floating-point types used in expressions.

arithmetic types

Integral, enumeration, and floating-point data types.

array

A set of elements with the same type.

ASCII (American Standard Code for Information Interchange)

A set of 256 codes that many computers use to represent letters, digits, special characters, and other symbols. Only the first 128 of these codes are standardized; the remaining 128 are special characters that are defined by the computer manufacturer.

associativity

Referring to operators, the precedence rules that apply when more than one operator is assigned to an operand. (For example, in the expression `*p++`, the indirection operator `*` is applied before the unary increment operator `++`.)

base name

The portion of the file name that precedes the file-name extension. For example, `samp` is the base name of the file `samp.c`.

batch file

A text file containing MS-DOS commands that can be invoked from the MS-DOS command line.

binary expression

An expression consisting of two operands joined by a binary operator.

binary operator

Operators used in binary expressions. Binary operators in the C language are the multiplicative operators (* /), additive operators (+ -), shift operators (<< >>), relational operators (< > <= >= == !=), bitwise operators (& | ^), logical operators (&& ||), and sequential-evaluation operator (,).

block

A sequence of declarations, definitions, and statements enclosed within curly braces ({}).

child process

A new process started by a currently running process.

CL

The command used by the Microsoft C Optimizing Compiler to compile and link programs.

compact memory model

A memory model that allows for more than one data segment and only one code segment.

complex declarator

A declaration containing more than one array, pointer, or function modifier.

constant expression

Any expression that evaluates to a constant and may involve integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant expressions.

declaration

A construct that associates the name and the attributes of a variable, function, or type.

declarator

An identifier that can be modified with brackets ([]), asterisks (*), or parentheses (()) to declare an array, pointer, or function type, respectively.

definition

A construct that initializes and allocates storage for a variable, or that specifies the name, formal parameters, body, and the return type of a function.

directive

An instruction to the C preprocessor to perform a specific action on source-program text before compilation.

emulator

A floating-point math package that provides software emulation of the operations of a math coprocessor.

enumeration set

The set of legal values defined for an enumeration type.

enumeration type

A user-defined data type that specifies a particular set of legal values.

environment table

The part of MS-DOS that stores environment variables and their values.

environment variable

A variable stored in the environment table that provides MS-DOS with information (where to find executable files and library files, where to create temporary files, etc.).

errorlevel code

See **exit code**.

escape sequence

A specific combination of a backslash (\) followed by a letter or combination of digits, which represents white-space and nongraphic characters within strings and character constants.

exit code

A code returned by a program to MS-DOS indicating whether or not the program ran successfully.

expression

A combination of operands and operators that yields a single value.

external level

The part of a C program outside of all function declarations.

file handle

A value returned by library functions that open or create files, used to refer to that file in later operations.

file pointer

A pointer that indicates the current position in an input or output stream. It is updated to reflect the new position each time a read or write operation takes place.

formal parameters

Variables that receive values passed to a function when the function is called.

forward declaration

A function declaration that establishes the attributes of a function so that it can be called before it is defined or called from a different source file.

function

A collection of declarations and statements returning a value that can be called by name.

function body

A compound statement containing the local variable declarations and statements of a function.

function call

An expression that passes control and actual arguments (if any) to a function.

function declaration

A declaration that establishes the name, return type, and storage class of a function that is defined explicitly elsewhere in the program.

function definition

A definition that specifies a function's name, its formal parameters, the declarations and statements that define what it does, and (optionally) its return type and storage class.

function prototype

A function declaration that includes a list of the names and types of formal parameters in the parentheses following the function name.

fundamental data types

A set of basic C data types, which includes all integer, character, floating-point, and enumeration types.

global

See **lifetime**; **visibility**.

heap

An area of memory set aside for dynamic allocation by a program.

huge memory model

A memory model that allows for more than one code segment and more than one data segment and that allows individual data items to span more than one segment.

include file

A text file that is merged into another text file using the **#include** preprocessor directive.

internal level

The parts of a C program within function declarations.

keyword

A word with a special, predefined meaning for the C compiler.

level

See **internal level** and **external level**.

large memory model

A memory model that allows for more than one segment of code and more than one segment of data, but with no individual data items spanning a single segment.

library

A file that stores related modules of compiled code. The linker extracts modules from the library and combines them with other program object modules to create executable program files.

lifetime

The period, during program execution, within which a variable or function exists. An item with a "local" lifetime (a "local item") has storage and a defined value only within the block where the item is defined or declared.

linked list

A data structure consisting of a list of entries, each of which includes a pointer to the next entry.

local

See **lifetime**; **visibility**.

loop optimization

Optimizations that reduce the amount of code executed for each loop iteration in a program.

low-level input and output routines

Run-time library routines that perform unbuffered, unformatted I/O operations.

lvalue

An expression (such as a variable name) that refers to a memory location and is required as the left-hand operand of an assignment operation or the single operand of a unary operator.

macro

An identifier defined in a **#define** preprocessor directive to represent another series of tokens.

manifest constant

An identifier defined in a **#define** preprocessor directive to represent a constant value.

medium memory model

A memory model that allows for more than one code segment and only one data segment.

member

One of the elements of a structure or union.

memory model

One of the models that specifies how memory is set up for program code and data. (See **small memory model**, **medium memory model**, **compact memory model**, **large memory model**, and **huge memory model** for descriptions of standard memory models.)

MS-DOS interface functions

Run-time library routines that provide access to MS-DOS interrupts and system calls.

multidimensional array

An array of arrays.

NAN

An abbreviation that stands for “not a number.” The 8087 or 80287 coprocessor generates NANs when the result of an operation cannot be represented in the IEEE format. For example, if you try to add two positive numbers whose sum is larger than the maximum value permitted by the processor, the coprocessor returns a NAN instead of the sum.

naming classes

Categories that the language sets up to distinguish between the identifiers used for different kinds of items.

new-line character

The character used to mark the end of a line of a text file, or the escape sequence (`\n`) used to represent this character. In MS-DOS “text mode,” carriage-return–line-feed (CR-LF) combinations are translated to into a single line-feed (LF) character on input, and line-feed characters are translated to carriage-return–line-feed combinations on output.

null character

The ASCII character encoded as the value 0, represented as the escape sequence (`\0`) in a source file.

null pointer

A pointer to nothing, expressed as the integer value 0.

object

A region of memory that can be examined. A modifiable object can also have a value stored into it (that is, it can be altered as well as examined).

object file

A file containing relocatable machine code, created as the result of compiling a source file.

operand

A constant or variable value that is manipulated in an expression.

operator

One or more symbols that specify how the operand or operands of an expression are manipulated.

overlay

Part of a program that is read into memory from disk only if and when it is needed.

parent process

A process that generates a child process using one of the **spawn**, **exec**, or **system** families of run-time library functions.

pass

One of the three stages of compilation (preprocessing/parsing, code generation, and optimization), or the executable file that performs one of these stages.

peephole optimization

Optimizations performed on a small part of the generated code.

pointers

A variable containing the address of another variable.

pragma

An instruction to the compiler to perform a particular action at compile time.

precedence

The relative position of an operator in the hierarchy that determines the order in which expressions are evaluated.

preprocessor

A text processor that manipulates the contents of a C source file during the first phase of compilation.

preprocessor directive

See **directive**.

process

A program being executed by MS-DOS.

prototype

See **function prototype**.

RAM disk

An area of memory that is used to load and save files in the same way as a disk drive but allows more rapid access to files than a disk drive. Unlike a disk drive, a RAM disk is not suitable for long-term storage because its contents are volatile: that is, they disappear if the machine is powered off.

relocatable

Not containing absolute addresses.

run time

The time during which a previously compiled and linked program is executing.

run-time library

A file containing the routines needed to implement certain functions of the Microsoft C language.

scalar types

In C, integral, enumerated, floating-point, and pointer types.

scope

The parts of a program in which an item can be referenced by name. The scope of an item may be limited to the file, function, block, or function prototype in which it appears.

segment

An area of memory, less than or equal to 64K, containing code or data.

sequence point

A point in a C program where all expressions lexically preceding the point are guaranteed to have been evaluated.

side effects

Changes in the state of objects that occur as a result of expression evaluation.

sizeof operator

A C operator that can be used to determine the amount of storage, in bytes, associated with an identifier or a type.

small memory model

A memory model that allows for only one code segment and only one data segment.

source file

A text file containing C-language code.

stack

A dynamically shrinking and expanding area of memory in which data items are stored in consecutive order and removed on a last-in, first-out basis.

stack probe

A short routine called on entry to a function to verify that there is enough room in the program stack to allocate local variables required by the function and, if so, to allocate those variables.

static

A storage class that allows variables keep their values even after the program exits the block in which the variable is declared.

stream functions

Run-time library functions that treat data files and data items as “streams” of individual characters.

string

An array of characters, terminated by a null character (`\0`).

string literal

A string of characters and escape sequences delimited by double quotes (`" "`). Every string literal has a type of “array of **char**.” An array of elements with **char** type.

structure

A set of elements, which may be of different types, grouped under a single name.

structure member

One of the elements of a structure.

subscript expression

An expression, usually used to reference array elements, representing an address that is offset from a specified base address by a given number of positions.

symbolic constant

See **manifest constant**.

tag

The name assigned to a structure, union, or enumeration type.

ternary expression

An expression consisting of three operands joined by the ternary (**? :**) operator, used to evaluate either of two expressions depending on the value of a third expression.

text mode

The file-processing mode in which carriage-return-line-feed combinations are converted to a single line-feed character on input and reconverted to carriage-return-line-feed combinations on output.

token

The most fundamental unit of a C source program that is meaningful to the compiler.

two's complement

A type of base-2 notation used to represent positive and negative numbers in which negative values are formed by complementing all bits and adding 1 to the results.

type

A description of a set of values; for example, a variable of type **int** can have any of a set of integer values within the range specified for the type on a particular machine.

type cast

An operation in which an operand of one type is converted to an operand of a different type.

type checking

An operation in which the compiler verifies that the operands of an operator are valid or that the actual arguments in a function call are of the same types as the corresponding formal parameters in the function definition and function prototype.

type declaration

A declaration that defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type.

typedef declaration

A declaration that defines a shorter or more meaningful name for an existing C data type or for a user-defined data type. Names defined in a **typedef** declaration are often referred to as “typedefs.”

type name

A specification of a particular data type that appears in variable declarations, in the formal-parameter lists of function prototypes, in type casts, and in **sizeof** operations.

unary expression

An expression consisting of a single operand preceded or followed by a unary operator.

unary operator

An operator that takes a single operand. Unary operators in the C language are the complement operators (**-** **~** **!**), indirection operator (*****), increment (**++**) and decrement (**--**) operators, address-of operator (**&**), and **sizeof** operator. The unary plus operator (**+**) is also implemented syntactically, but has no semantics associated with it.

union

A set of values of different types that occupy the same storage space.

unresolved reference

A reference to a global or external variable or function that cannot be found, either in the modules being linked or in the libraries that are linked with those modules.

usual arithmetic conversions

Type conversions performed by the Microsoft C Optimizing Compiler on operands of integral or floating-point types in an expression to bring the operands to a common type.

visibility

The characteristic of a variable or function that describes the parts of the program in which it can be referenced by name. An item has global visibility if it is visible in all source files constituting the program and local visibility in a single source file otherwise.

white-space character

Characters that delimit items in a C source program, including space, tab, line-feed, carriage-return, form-feed, vertical-tab, and new-line characters.

wild card

One of the MS-DOS characters (?) and *) that can be expanded into one or more characters in file-name references.

USER'S GUIDE INDEX

- * (asterisk), wild-card character, 130
- | (bar), 10
- { } (braces), 10
- [] (brackets), 9
- / (forward slash) option character
 - CL, 53
 - linker, 119
- (hyphen) option character, CL, 53
- ? (question mark), wild-card character, 130
- _ (underscore), in names, 61, 73

- 80186/80188 processor, 81
- 80286 processor, 81
- 8087/80287
 - coprocessor
 - exceptions, 175
 - math package, 164
 - suppressing use of, 174
 - library, 27
- 87.LIB, 27

- /A option, 152, 153, 154
- Abstract declarator, defined, 287
- /AC option, 54, 141
- Address space, 217
- Addresses
 - components, 138
 - far, 138
 - huge, 138
 - near, 138
- Aggregate types, defined, 287
- /AH option, 54, 143
- /AL option, 54, 142
- Alias checking, 89
- Alignment. *See* Storage alignment
- alloc_ text pragma, 159
- Alternate math library, 169
- /AM option, 54, 141
- argv variable, 30, 42, 128
- Arguments
 - argument-type list, defined, 287
 - command line, 131
 - linker options, 119
 - listing options, 62
 - macros, 280
 - main function. *See* main function
 - variable number of, 106, 183, 221
- Arguments (*continued*)
 - wild card, on command line, 130
- Argument-type list, 86
- argv variable, 30, 42, 128
- /AS option, 54, 140
- Assembly-listing files
 - creating, 61
 - extensions, 62
 - format, 71
- Asterisk (*), wild-card character, 31, 130
- AUTOEXEC.BAT file, 26
- AUX (device name), 64
- /Aw option, 109

- Back-up procedures, 15
- Bar (|), 10
- Base name, defined, 288
- /BATCH (/B) linker option, 121
- Batch files, 191
- BEGDATA class name, 124
- Bibliography, 11
- \BIN subdirectory, 24
- \BIN\SAMPLE subdirectory, 25, 26
- Binary mode, 31, 111
- BINMODE.OBJ, 25, 31, 111
- Bit fields, 215
- Bold font, 8
- Braces { }, 10
- Brackets [], 9
- BSS class name, 124
- Buffers parameter (CONFIG.SYS), 16, 26, 37
- Byte length, 212
- Byte order, 214, 225

- /c option, 40, 57
- /C option, 80
- C1.EXE file, 29
- C2.EXE file, 29
- C3.EXE file, 29
- Calling conventions
 - C, 106, 183
 - controlling
 - cdecl keyword, 107
 - fortran and pascal keywords, 107
 - /Gc option, 107
- FORTRAN/Pascal, 106, 183

- Capital letters
 - small, 10
 - use of, 8
- Carriage-return-line-feed (CR-LF)
 - translation, 111
- Case significance
 - linker, 120, 123
- cdecl keyword
 - defined, 107
 - /Gc option, used with, 184
 - include files, used in, 100
 - /Za option, used with, 99
- char type, changing default, 105
- Character
 - classification, macros, 218
 - set, 217
 - types
 - signed, 218
 - unsigned, 218
- check_stack pragma, 97, 106, 183
- CL command
 - defined, 289
 - exit codes, 192
 - file processing, 48
 - format, 48
 - path specifications, 50
 - stopping, 51
- CL environment variable, 51
- CL options
 - 80186/80188 or 80286 processors,
 - using, 81
 - /A, 152, 153, 154
 - /AC, 54, 141
 - /AH, 54, 143
 - /AL, 54, 142
 - /AM, 54, 141
 - /AS, 54, 140
 - assembly listing, 61
 - /Aw, 109
 - /c, 40, 57
 - /C, 80
 - case sensitivity of, 50, 53
 - command line, order, 54
 - comments, preserving, 80
 - constants and macros, defining, 75
 - /D, 75
 - data segments, naming, 157, 159, 185
 - data threshold, setting, 156
 - debugging with CodeView debugger,
 - 41, 87
 - debugging with SYMDEB debugger,
 - 87
 - default char type, changing, 105
 - default libraries, 55
 - differences from linker options, 120
 - /E, 79

- CL options (*continued*)
 - /EP, 79
 - external names, restricting length of,
 - 103
 - /F, 123
 - /Fa, 61, 71
 - /Fc, 61
 - /Fe, 60
 - /Fl, 61
 - floating point
 - coprocessor, maximum efficiency
 - with, 168
 - coprocessor, maximum efficiency
 - without, 169
 - coprocessor, maximum precision
 - with, 168
 - coprocessor, maximum precision
 - without, 167
 - default, 167
 - default libraries, 54, 165
 - effects, 165
 - flexibility, maximum, 172, 174
 - function calls, 168, 169
 - in-line instructions, 167, 168, 169,
 - 170
 - library use, controlling, 170
 - /Fm, 61
 - /Fo, 58
 - format, 53
 - FORTRAN/Pascal, calling
 - convention, 107
 - /FPa, 55, 165, 169
 - /FPc, 55, 165, 168
 - /FPc87, 55, 165, 169
 - /FPI, 55, 165, 167
 - /FPI87, 55, 165, 168
 - /Fs, 41, 61
 - function declarations, generating, 86
 - /G0, 81
 - /G1, 81
 - /G2, 81
 - /Gc, 107
 - /Gs, 97, 106, 183
 - /Gt, 156
 - /Gw, 109
 - /H, 103
 - /HELP, 39, 56
 - /I, 80
 - include files, searching for, 80
 - /J, 105
 - language extensions, disabling, 99
 - line numbers, 87
 - line width, 65
 - /link, 48, 115
 - linker information, passing, 115
 - listing, 39, 56

CL options (*continued*)

- maximum optimization, 89
- memory models
 - code-pointer size, 153
 - compact, 141
 - data-pointer size, 153
 - default libraries, 54
 - huge, 143
 - large, 142
 - medium, 141
 - mixed, 152, 153, 154
 - segments, setting up, 109, 154
 - small, 140
- naming
 - executable files, 60
 - modules, 157
 - object files, 58
- /ND, 157, 159, 185
- /NM, 157
- /NT, 157
- /Oa, 89, 182
- object files
 - labeling, 103
 - naming, 58
 - specifying, 48
- object listing, 61
- /Od, 41, 87, 93
- /Oi, 89, 93, 181
- /Ol, 89, 94, 182
- /Op, 95
- optimization
 - alias checking, relaxing, 89, 182
 - code size, 89, 96
 - disabling, 41, 87, 93
 - execution time, 89, 96, 181
 - floating-point results, consistent, 95
 - intrinsic functions, 89, 93, 181
 - loops, 89, 94, 182
 - maximum, 96
 - /Oi, 181
 - program speed, 181
- option character
 - forward slash (/), 53
 - hyphen (-), 53
- /Os, 96
- /Ot, 96, 181
- /Ox, 89, 96, 106
- /P, 79
- page length, 65
- predefined identifiers, removing
 - definitions of, 78
- preprocessed listing, 79
- preprocessor
 - /c, 80
 - /D, 75

CL options (*continued*)

- preprocessor (*continued*)
 - /U and /u, 78
- /qc, 57
- QuickC
 - debugging interrupts, 58
 - null pointer, checking, 58
- /qc, 57
- /Zq, 58
- /Zr, 58
- /Sl, 65
- source files, specifying, 48, 49, 56
- source listing, 41, 61
- source/object listing, 61
- /Sp, 65
- special keywords, disabling, 145
- /Ss, 41, 66
- /St, 41, 66
- stack probes, removing, 97, 183
- standard places, ignoring, 81
- structure members, packing, 100
- subtitle, 41, 66
- suppressing
 - library selection, 104, 172
 - linking, 40, 57
- syntax checking, 85
- /Tc, 49, 56
- text segments, naming, 157
- titles, 41, 66
- /U and /u, 78
- /V, 103
- Version 4.0, new for, 206
- Version 5.0, new for, 198
- /W0, /W1, /W2, and /W3, 84
- warning level, 84
- /X, 80, 81
- XENIX compatible, 109
- /Za, 99, 145
- /Zd, 87, 123
- /Zg, 86
- /Zi, 41, 87, 123
- /Zl, 104, 172
- /Zp, 100
- /Zs, 85
- Class names
 - BEGDATA, 124
 - BSS, 124
 - CODE, 124
 - STACK, 124
- CL.ERR file, 30
- CL.EXE file, 29, 33
- CL.HLP file, 30
- CODE class name, 124
- Code pointers, mixed memory models, 153
- Code size, optimization, 89, 96

- Code-helper library, 27
- /CODEVIEW (/CO) linker option, 123
- CodeView debugger
 - CL option for, 41, 87
 - executable file for, 28
 - help file for, 30
 - linker option for, 123
- Command line
 - arguments
 - executable file, 127
 - maximum length, 128
 - stored program header, 128
 - suppressing processing of, 131
 - wild cards in, 130
 - CL, 48
 - error messages, 237
 - length, maximum, 48, 286
- Commands
 - CL, defined. *See* CL command
 - MS-DOS
 - PATH, 32, 36
 - SET, 32, 36
 - notational conventions, 8
- Comments, preserving, 80
- Compact memory models. *See* Memory models
- Compatibility
 - floating-point options, 173
 - mLIBC7.LIB, 175
 - mLIBCE.LIB, 175
 - XENIX options, 109
- Compilation
 - conditional, 100
 - error messages, 244
- Compiler
 - differences, other compilers
 - portability problems, 218
 - differences, Version 4.0
 - CL options, 206
 - enhancements and additions, 203
 - language changes, 204
 - new library functions, 207
 - differences, Version 5.0
 - enhancements and additions, 195
 - language changes, 196
 - new CL options, 198
 - new library functions, 199
 - pragmas, new, 198
 - documentation, 4
 - error messages. *See* Error messages, compiler
 - files, default directory, 24
 - limits, 280
 - naming conventions, 72
 - passes, 29
 - stopping, 48, 51

- Compiler, converting from previous versions. *See* Compiler differences
- Compiler guide, organization, 4
- Compiler options. *See* CL options
- CON (device name), 64
- Conditional compilation, 76, 100
- CONFIG.SYS file, 26, 37
- Constants
 - defining, 75
 - manifest. *See* Constants, symbolic
 - size, maximum, 280
 - symbolic, 75
- Controlling
 - binary and text modes, 111
 - linker, 119
 - preprocessor, 78
 - segments, 122
 - stack size, 123
- Conventions, notational, 8
- Conversion
 - near pointers to long integers, 205
 - pointer arguments, 150
- Coprocessor, 8087/80287
 - exceptions, 175
 - math package for, 164
 - suppressing use of, 175
- Correctable error messages, 244
- /CPARMAXALLOC (/CP) linker
 - option, 122
- CR-LF (carriage-return-line-feed)
 - translation, 111
- CRT0.OBJ. *See* Start-up routine
- ctype.h macros, 218
- Customized memory models. *See* Mixed memory models
- CV.ERR file, 30
- CV.EXE file, 29
- CV.HLP file, 30

- /D option, 75
- Data
 - files
 - binary, 31
 - text, 31
 - passing to programs, 127
 - portability, 223
 - segment
 - data threshold, setting, 156
 - default, contents, 156
 - default name, 157
 - mixed memory models, 154
 - naming, 157
 - types, size of, 212
- Data pointers, mixed memory models, 153

- _DATA segment, 157
- Data threshold, setting, 156
- Debugging, preparing for
 - /CODEVIEW linker option, 123
 - /Zi, /Zd, and /Od options, 87
- Declarations, maximum level of
 - nesting, 280
- Default libraries
 - See also* Libraries, default
 - object files, used in, 117
 - suppressing selection, 104, 172
- DEMO.C program, 31, 41
- Denormal numbers, 164, 284
- Device names, 64
- DGROUP group, 124, 157
- Differences from previous versions. *See*
 - Compiler differences
- Directory names, notational
 - conventions, 8
- Disks
 - backing up, 15
 - contents, 15
- Distribution disk, Master, 16, 17
- Documentation, compiler, 4
- /DOSSEG (/DO) linker option, 123
- DS register, 154
- /E option, 79
- Ellipses, use of, 9
- EM.LIB, 27
- EMOEM.ASM
 - file, 176
 - program, 31
- Emulator
 - described, 163
 - function calls, 168
 - in-line instructions, 167
 - library, 27, 167, 168
- environ variable, 129
- Environment
 - changing, 37
 - portability problems, 222
 - table
 - pointer to, 129
 - size, increasing, 37
 - size, maximum, 286
 - suppressing processing of, 131
 - variable names, notational
 - conventions, 8
 - variables
 - CL, 35, 51
 - default settings, 26
 - defined, 32
 - defining, 36
 - INCLUDE, 33, 35, 80
- Environment (*continued*)
 - variables (*continued*)
 - LIB, 33, 35, 117
 - NEW-VARS.BAT, 26
 - overriding, 37
 - PATH, 33, 34, 127
 - SET, 127
 - SETUP, 32
 - TMP, 33, 35
 - using, 33
 - envp variable, 30, 42, 129
 - /EP option, 79
 - Error messages
 - compiler
 - command line, 237
 - compilation, 244
 - correctable, 244
 - fatal, 243, 244
 - identifying, 82
 - redirecting, 83
 - warning, 244, 269
 - floating-point exceptions, 284
 - format. *See* Error messages, compiler
 - run time, 281
 - run-time library, 281
 - source listings, 67
 - warning messages, setting level of, 84
- Errorlevel codes. *See* Exit codes
- ERROUT.EXE file, 29
- Evaluation order, 221
- Exception, 284
- exec function, 127, 131
- Executable files
 - CL command and, 50
 - command-line arguments, 127
 - compiler and utilities, 28
 - extensions, 28, 60
 - invoking, 28
 - naming, default, 60
 - naming with CL, 60
 - packing, 121
 - passing data to, 127
 - running, 127
 - search path, 33
- Execution-time optimization, 89, 96, 181
- EXEMOD.EXE file, 29
- /EXEPACK (/E) linker option, 121
- EXEPACK.EXE file, 29
- Exit code, 131, 191
- Extensions
 - executable files, 60
 - listing files, defaults for, 62
 - map files, 62
 - object files, 59
 - object-listing files, 62

Extensions (*continued*)

- source-listing files, 62
- source/object-listing files, 62

External names, 103

/F option, 123

/Fa option, 61, 71

far keyword

- default addressing conventions, 144
- effects

- data declarations, 146, 184

- function declarations, 148

- library routines, used with, 145

- small-model programs, used in, 140

- /Za option, used with, 99

Far pointers, 144

/FARCALLTRANSLATION (/F)

- linker option, 122, 185

Fatal-error messages, 243, 244

/Fc option, 61

/Fe option, 60

File names

- notational conventions, 8
- uppercase and lowercase letters, using, 50

Files

- assembly listing, 61, 71

- AUTOEXEC.BAT, 26

- C1.EXE, 28

- C2.EXE, 28

- C3.EXE, 28

- CL.EXE, 33

- compiler, 24

- CONFIG.SYS, 26, 37

- data. *See* Data files

- executable. *See* Executable files

- include. *See* Include files

- library, 30

- listing, preprocessed, 79

- locating, 32

- map

- creating, 61, 64, 122, 123

- default names, 62

- listing formats, 73

- /MAP linker option, 123

- number open, maximum, 286

- object

- See also* Object files

- CL command, used with, 48, 50

- defined, 294

- listing, 61, 62, 71

- parameter (CONFIG.SYS), 16, 26, 37

- size, maximum, 286

- source, 48, 297

- source listing. *See* Source-listing files

Files (*continued*)

- source/object listing. *See*

- Source/object-listing files

- temporary

- space requirements, 280

- TMP, 35

/Fl option, 61

Floating point

- not loaded, 282

- operations

- error messages, 284

- floating-point exceptions, 284

- optimizing for consistency in, 95

- options

- compatibility, 173

- coprocessor, maximum efficiency with, 168

- coprocessor, maximum efficiency without, 169

- coprocessor, maximum precision with, 168

- coprocessor, maximum precision without, 167

- default, 167

- default libraries, 55, 165

- effects, 165

- flexibility, maximum, 172, 174

- function calls, 168, 169

- in-line instructions, 167, 168, 169, 170

- library, controlling use, 170

- listed, 165

- selecting, 54

/Fm option, 61

/Fo option, 58

fortran keyword, 99, 107, 184

Forward slash (/)

- CL option character, 53

- linker option character, 119

/FPa option, 55, 165, 169

/FPc option, 55, 165, 168

/FPc87 option, 55, 165, 169

/FPi option, 55, 165, 167

/FPi87 option, 55, 165, 168

/Fs option, 41, 61

function pragma, 93

Functions

- arguments, variable number of, 106, 183, 221

- calling conventions

- C, 106, 183

- FORTTRAN/Pascal, 106, 183

- declarations

- generating, 86

- near and far keywords, 148

- /G0 option, 81
- /G1 option, 81
- /G2 option, 81
- /Gc option, 107
- getenv function, 129
- Global symbols. *See* Public symbols
- /Gs option, 97, 106, 183
- /Gt option, 156
- /Gw option, 109

- /H option, 103
- Heap, 30, 105
- /HELP option
 - CL, 39, 56
 - linker, 120
- /help option. *See* /HELP option, CL
- Huge arrays, 143
- huge keyword
 - data declarations, effects in, 146, 184
 - default addressing conventions, 144
 - library routines, used with, 145
 - small-model programs, used in, 140, 99
- Huge memory model. *See* Memory models
- Huge pointers, 144
- Hyphen (-), CL option character, 53

- /I option, 80
- Identifier length. *See* Names, length
- Identifiers
 - length, maximum, 280
 - predefined
 - listed, 77
 - M_I86, 77
 - M_I86xM, 77
 - MS-DOS, 77
 - NO_EXT_KEYS, 78, 100
 - removing definitions of, 78
- IF ERRORLEVEL (MS-DOS command), 132
- # include directive, 29
- Include files
 - compiler, provided with, 29
 - directory specification, 80
 - nesting, maximum level of, 280
 - portability problems, 212
 - search path, 33, 80, 81
 - standard places, 35
- \INCLUDE subdirectory, 24
- \INCLUDE\SYS subdirectory, 24, 29
- INCLUDE variable
 - defined, 35
 - overriding, 80, 81

- Inexact, 284
- Infinites, 164
- /INFORMATION (/I) linker option, 121
- In-line instructions, 167, 168
- Instruction sets
 - 80186/80188 processor, 81
 - 80286 processor, 81
 - 8086/8088 processor, 81
- intrinsic pragma, 93
- Italics, 9

- /J option, 105
- Kernighan, Brian W., 11
- Key sequences, notational conventions, 10
- Keywords
 - cdecl, 99, 107, 184
 - defined, 292
 - far. *See* far keyword
 - fortran, 99, 184
 - huge. *See* huge keyword
 - near. *See* near keyword
 - pascal, 99, 184
 - special, 99
 - Version 4.0, new for, 206

- Language extensions
 - disabling, 99
 - listed, 99
- Large memory model. *See* Memory models, large
- LIB library manager, 130
- LIB variable, 33, 117
- LIB.EXE file, 29
- mLIBC7.LIB, 168, 169
- mLIBCA.LIB, 168, 169
- mLIBCE.LIB, 167, 168, 175
- mLIBFA.LIB, 27
- mLIBFP.LIB, 27
- LIBH.LIB, 27
- Libraries
 - 8087/80287 package, 168, 169
 - alternate math, 168, 169
 - controlling use, 170, 171
 - creating
 - /FPc, compiling modules with, 169
 - /Zl, compiling modules with, 104, 172
 - default
 - See also* Default libraries
 - directory, 24
 - /FP and /A options, 55, 115
 - ignoring, 118, 121

- Libraries (*continued*)
 - default (*continued*)
 - overriding, 117
 - suppressing selection, 104
 - defined, 292
 - emulator, 167, 168, 175
 - mLIBC7.LIB, 168, 169
 - mLIBCA.LIB, 40, 168, 169
 - mLIBCE.LIB, 167, 168, 175
 - mixed-model programs, 155
 - names in object files, 115, 165
 - notational conventions, 25
 - RAM disk, used with, 38
 - run time, defined, 296
 - search
 - order, 171
 - path, 33, 117
- SETUP
 - math packages, choosing, 20
 - memory models, choosing, 18
 - naming conventions, 23
- specifying, 116
- standard, 56
- standard places, 35, 117
- uncombined
 - 8087/80287 floating point, 27
 - 87.LIB, 27
 - alternate math, 27, 40
 - code helper, 27
 - corresponding combined libraries, 27
 - EM.LIB, 27
 - emulator, 27
 - floating point, 27
 - mLIBFA.LIB, 27
 - mLIBFP.LIB, 27
 - LIBH.LIB, 27
 - standard, 27
 - using, reasons for, 26
- Library
 - manager, 29
 - routines
 - exec, 127, 131, 230
 - getenv, 129
 - intrinsic forms, 93
 - MS-DOS dependent, 229
 - putenv, 129
 - setmode, 111
 - spawn, 127, 131, 230
 - syntax, changes, 208
 - system, 127
 - Version 4.0, new for, 207
 - Version 5.0, changed for, 201
 - Version 5.0, new for, 199
- \LIB subdirectory, 24
- Limits
 - compiler, 280
 - run time, 286
- Line width, source listings, 65
- /LINENUMBERS (/LI) linker option, 123
- /link option, 48, 115
- Linker
 - error messages, 82
 - executable file for, 29
- Linker options
 - abbreviations, 119, 120
 - /B (/BATCH), 121
 - case sensitivity, 120, 123
 - CL options, differences from, 120
 - /CODEVIEW (/CO), 123
 - command line, order on, 120
 - /CPARMAXALLOC (/CP), 122
 - debugging with CodeView debugger, 123
 - default libraries, ignoring, 118, 121
 - displaying, 120
 - /DOSSEG (/DO), 123
 - executable files, packing, 121
 - /EXEPACK (/E), 121
 - /FARCALLTRANSLATION (/F), 122, 185
 - /HELP (/HE), 120
 - /INFORMATION (/I), 121
 - line numbers, displaying, 123
 - /LINENUMBERS (/LI), 123
 - map file, 123
 - /MAP (/M), 123
 - /NODEFAULTLIBRARYSEARCH (/NOD)
 - avoiding ambiguity in library
 - customized memory models, 155
 - defined, 121
 - overriding default libraries, 1
 - /NOFARCALLTRANSLATION (/NOF), 122
 - /NOIGNORECASE (/NOI), 123
 - /NOPACKCODE (/NOP), 122
 - numerical arguments, 119
 - optimizing intrasegment far calls, 122
 - /PACKCODE (/PAC), 122, 186
 - packing code segments, 122
 - packing contiguous segments, 186
 - paragraph space, allocating, 122
 - /PAUSE (/P), 121
 - pausing, 121
 - process information, displaying, 121
 - Quick library, creating, 121
 - /QUICKLIB (/QU), 121
 - rules, 119

Linker options (*continued*)

- segments
 - number of, 122
 - ordering, 123
 - /SEGMENTS (/SE), 122
 - stack size, setting, 102, 123
 - /STACK (/ST), 102, 123
 - suppressing prompting, 121
 - translating far calls, 185
- LINK.EXE file, 29
- Listing CL options, 39, 56
- Listing files
 - assembly, 61, 71
 - map, 61
 - object, 61, 71
 - preprocessed, 79
 - source, 61, 67
 - source/object, 61, 72
- Long pointers. *See* Far pointers
- Loop optimization, 94, 182
- loop_opt pragma, 89, 94, 182

Macro definitions, 280

Macros

- arguments, maximum number, 280
- character classification, 218
- defined, 75
- notational conventions, 8
- main function
 - arguments to, 127
 - exit codes, 131

Map files

- creating, 61, 64, 123
- extensions, 62, 123
- /Fm option, 64
- format, 73
- /MAP linker option, 123
- program entry point, 74
- segment lists, 73
- symbol tables, 74
- /MAP linker option, 123

Math packages

- 8087/80287 package, 164
- alternate math, 164
- emulator, 163

Medium memory model. *See* Memory modelsMemory addresses. *See* Addresses

Memory allocation, stack, 30

Memory models

- CL options, 54
- compact, 54, 141, 289
- default, 137, 140, 167
- huge, 54, 143, 292
- large, 54, 142, 292

Memory models (*continued*)

- medium, 54, 141, 293
- mixed. *See* Mixed memory models
- notational conventions for files, 25
- options
 - code-pointer size, 153
 - compact model, 141
 - data-pointer size, 153
 - default libraries, 55
 - huge model, 143
 - large model, 142
 - medium model, 141
 - segment setup, 154
 - small model, 140
- small, 54, 137, 140
- standard
 - advantages, 139
 - common features, 140
 - disadvantages, 139
 - Version 4.0, new for, 206
- Memory models, customized. *See* Mixed memory models
- Memory-based disk emulator. *See* RAM disk
- ML I86 identifier, 77
- ML I86xM identifier, 77
- Mixed memory models
 - code pointers, 153
 - creating, 152
 - data pointers, 153
 - library support, 155
 - near, far, huge keywords, 144
 - segment setup options, 154
- Modules, naming, 157
- MS-DOS commands
 - IF ERRORLEVEL, 132
 - PATH, 32
 - SET, 32
- MS-DOS, identifier, 77

Names

- conventions, 108
- devices, 64
- executable files, 60
- external, 103
- global, 73
- length, 219
- modules, changing, 157
- object files, 58
- segments, changing, 157
- underscores (_), using in, 61, 73
- Naming conventions
 - compiler, 72
 - segments, 158

- NAN (not a number)
 - alternate math package, used with, 164
 - defined, 294
- /ND option, 157, 159, 185
- near keyword
 - data declarations, effects in, 146, 184
 - default addressing conventions, 144
 - function declarations, effects in, 148
 - library routines, used with, 145
- Near pointer, 144
- Nesting
 - declarations, 280
 - include files, 280
 - preprocessor directives, 280
- NEW-CONF.SYS file, 16, 26
- NEW-VARS.BAT file, 16, 26
- /NM option, 157
- NO87 variable, 174
- /NODEFAULTLIBRARYSEARCH
 - (/NOD) linker option
 - customized memory models, 155
 - default libraries, overriding, 1
 - defined, 121
- NO_EXT_KEYS, 77, 100
- /NOFARCALLTRANSLATION
 - (/NOF) linker option, 122
- /NOIGNORECASE (/NOI) linker option, 123
- /NOPACKCODE (/NOP) linker option, 122
- Not a number (NAN)
 - alternate math package, used with, 164
 - defined, 294
- Notational conventions, 8
- /NT option, 157
- NUL (device name), 64
- Null pointer
 - assignment, 132
 - checks, suppressing, 132
- NULL segment, 132, 282
- _nullcheck library routine, 133
- Null-pointer assignment, 282
- /O (optimization) options, 89
- /Oa option, CL, 89, 181

- Object files
 - See also* Files, object
 - CL command, 48, 50
 - default extension, 49, 56
 - defined, 294
 - extensions, 59
 - labeling, 103
 - library names in, 115, 165
- Object files (*continued*)
 - naming, 58
 - specifying to CL, 48
- Object listing. *See* Object-listing files
- Object-listing files
 - creating, 61
 - extensions, 62
 - format, 71
- /Od option, 41, 87
- /Oi option, 89, 181
- /Ol option, 89, 94, 182
- /Op option, 95
- Optimization
 - alias checking, relaxing, 89, 182
 - code size, 89, 96
 - consistent floating-point results, 89, 95
 - default, 47, 96
 - disabling, 87, 89, 93
 - execution time, 89, 181
 - /FPc87 option, effects of, 169
 - intrinsic functions, 93
 - intrinsic pragmas, 181
 - listing files, 63
 - loops, 94, 182
 - maximum, 89, 96
 - options, 88
 - stack probes, removing, 97, 183
- Optimizing. *See* Optimization
- Optional fields, notational conventions, 9
- Options, CL. *See* CL options
- Options, linker. *See* Linker options
- /Os option, 96
- /Ot option, 96, 181
- Overlays, 116, 295
- Overview, 3
- /Ox option, 89, 96, 106

- /P option, 79
- pack pragma, 100
- /PACKCODE (/PAC) linker option, 122, 186
- Packing
 - executable files, 121
 - structure members, 100
- PACKING.LST file, 16
- Page length, source listings, 65
- Paragraph space, 122
- pascal keyword, 99, 107, 184
- PATH command, 32, 36
- Path names
 - CL command line, 50
 - notational conventions, 8
 - portability problems, 212

PATH variable, 33, 36, 127
 /PAUSE (/P) linker option, 121
 Placeholders, 9
 Pointers
 arguments, size conversion, 150
 code, 153
 far, 144, 153
 huge, 144
 manipulation, 216
 near
 conversion to long integers, 205
 customized memory models, 153
 near keywords, used with, 144
 subtracting in huge-model programs, 143
 Portability
 address space, 217
 bit fields, 215
 byte length, 212
 byte order, 214, 225
 case distinctions, 219
 character set, 217
 data, 223
 data types, size of, 212
 environment differences, 222
 evaluation order, 221
 functions with variable number of
 arguments, 221
 guidelines, 212
 hardware, 212
 identifier length, 219
 include files, 212
 path names, 212
 pointer manipulation, 216
 register variables, 219
 shift operations, 218
 side effects, 221
 sign extension, 218
 signed and unsigned char types, 218
 storage alignment, 213
 type conversion, 220
 word length, 212
 Practice session, 41
 Pragas
 alloc_text, 159
 check_stack, 97, 106, 183
 function, 93
 intrinsic, 93
 loop_opt, 89, 94, 182
 pack, 100
 same_seg, 159, 185
 Version 4.0, new for, 206
 Version 5.0, new for, 198
 Preprocessor
 macro arguments, maximum number
 of, 280

Preprocessor (*continued*)
 macro definitions, maximum size of, 280
 nesting, maximum level of, 280
 options
 comments, preserving, 80
 /D, 75
 predefined identifiers, removing
 definitions of, 78
 use, 75
 PRN (device name), 64
 Processors
 80186/80188, using, 81
 80286, using, 81
 8086/8088, using, 81
 Program header, 128
 Prompts, 10
 Public names. *See* External names;
 Public symbols
 Public symbols, listing, 64, 123
 putenv function, 129

 /qc option, 57
 /QU (/QUICKLIB) linker option, 121
 Question mark (?), wild-card character, 31, 130
 QuickC. *See* CL options, QuickC
 Quotation marks, use of, 10

 RAM disk
 advantages, 38
 libraries, used for, 38
 temporary files, used for, 33, 35, 38
 Register variables, 179, 219
 Relocatable, defined, 296
 Return codes. *See* Exit Codes
 Ritchie, Dennis M., 11
 Run file. *See* Executable file
 Run time
 error messages, 281
 limits, 286

 same_seg pragma, 159, 185
 Search paths
 changing
 CL options, using, 37
 include files, 81
 libraries, 117
 executable files, 33
 include files, 33, 35, 80
 libraries, 33, 117
 standard, 32
 temporary files, 35

- Segment lists
 - map files, 73
 - source listings, 71
- Segments
 - data
 - default name, 157
 - mixed memory models, 154
 - names, 157
 - naming, 157
 - threshold, effect of, 156
 - default, 137
 - defined, 137
 - names, changing, 157
 - naming conventions, 158
 - NULL, 132, 282
 - number allowed, 122
 - order, 123
 - setting up, 109, 154
 - source listing, 71
 - stack, 154
 - text
 - default name, 157
 - naming, 157
- /SEGMENTS (/SE) linker option, 122
- SET command, 32, 36
- SET variable, 127
- _setargv library routine, 131
- SETARGV.OBJ file, 25, 130
- SETENV utility, 37
- _setenvp routine, 131
- setmode function, 111
- SETUP
 - arguments, 18
 - default file organization, 24
 - disk, 16
 - installation directories, choosing, 18
 - libraries, naming, 23
 - math packages, choosing, 20
 - memory models, choosing, 18
 - operations, 16
 - PACKING.LST file, 16
 - running, 17
- Shift operations, 218
- Short pointers. *See* Near pointers
- Side effects, 221
- Sign extension, 218
- Signed char type, 218
- sizeof operator, 143
- /Sl option, 65
- Small capitals, use of, 10
- Small memory model. *See* Memory models
- Source files
 - default extension, 49, 56
 - defined, 297
 - specifying to CL, 48
- Source listing. *See* Source-listing files
- Source-listing files
 - creating, 61
 - described, 61
 - error messages, 67
 - extensions, 62
 - format, 67, 68
 - line width, 65
 - page length, 65
 - segment lists, 71
 - subtitles, 66
 - symbol tables, 69
 - titles, 66
- Source/object-listing files
 - creating, 61
 - extensions, 62
 - format, 72
 - /Sp option, 65
- spawn function, 127, 131
- Special keywords, disabling, 145
- \SRC subdirectory, 25
- /Ss option, 41, 66
- SS register, 154
- /St option, 41, 66
- Stack
 - defined, 297
 - fixed, 105
 - memory allocation from, 30
 - overflow, 281
 - probes, 97, 183, 297
 - segments, mixed memory models, 154
 - size
 - default for C programs, 102
 - setting, 102, 123
- STACK class name, 124
- /STACK (/ST) linker option, 102, 123
- Standard places
 - changing, 81
 - defined, 32
 - ignoring, 81
 - include files, 35
 - libraries, 35, 117
 - temporary files, 35
- Start-up
 - routine, 30, 231
 - source files, 32
- stdargv module, 130
- Storage alignment, 213
- Strings
 - length, maximum, 280
 - notational conventions, 10
- Structures, packing, 100
- Subdirectories
 - \BIN, 24
 - \BIN\ SAMPLE, 25, 26
 - \INCLUDE, 24

Subdirectories (*continued*)
 \INCLUDE\SYS, 24, 29
 \LIB, 24
 \SRC, 25

Subtitles, source listings, 66

Switches. *See* Options

Symbol tables

map files, used in, 74

object files, used in (/Zi option), 87

source listings, used in, 69

SYMDEB debugger, CL option for, 87

Syntax conventions. *See* Notational conventions

system function, 127

System-level definitions, 29

/Tc option, 48, 49, 56

Temporary files

compiler, after stopping, 51

default directory, 24

RAM disk, used for, 38

removing, 26

standard places, 35

Text mode, 30, 111

_TEXT segment, 157

Text segments

default name, 157

naming, 157

Titles, source listings, 66

TMP variable, 33, 35

Two's complement, defined, 298

Types

checking, 86

conversion, 220

/U and /u options, 78

Underflow, 284

Underscore (_) in names, 61, 73

Unsigned char type, 218

Uppercase letters, use of, 8, 50

Utilities

default directory, 24

ERROUT. *See* ERROUT.EXE

EXEMOD. *See* EXEMOD.EXE

EXEPACK. *See* EXEPACK.EXE

LIB. *See* LIB.EXE

LINK. *See* LINK.EXE

/V option, 103

Variables, environment, 32

See also Environment variables

Variables, register. *See* Register variables

mVARSTCK.OBJ file, 30, 105

Vertical bar (|), 10

/W0, /W1, /W2, and /W3 options, 84

Warning error messages, 84, 244, 269

Wild card

arguments, 31, 130

characters, 56

Windows applications

/Aw option, 109

/Gw option, 109

/X option, 80, 81

XENIX-compatible options, 109

/Za option, 99, 145

/Zd option, 87, 123

/Zg option, 86

/Zi option, 41, 87, 123

/Zl option, 104, 172

/Zp option, 100

/Zq option, 58

/Zr option, 58

/Zs option, 85